Adaptive and Intelligent Memory Systems

Aswinkumar Sridharan

ANNÉE 2016

UNIVERSITÉ DE RENNES 1

UNIVERSITE BRETAGNE LOIRE

THÈSE / UNIVERSITÉ DE RENNES 1
sous le sceau de l'Université Européenne de Bretagne

pour le grade de

DOCTEUR DE L'UNIVERSITÉ DE RENNES 1

Mention : Informatique

École doctorale Matisse

présentée par

Aswinkumar SRIDHARAN

préparée à l'unité de recherche INRIA
Institut National de Recherche en Informatique et Automatique –
Université de Rennes 1

Adaptive and Intelligent Memory Systems

Thèse soutenue à Rennes
le 15 Décembre 2016

devant le jury composé de :

Lionel LACASSAGNE
Professeur à University Pierre et Marie Curie / *Président*

Pierre BOULET
Professeur à l'Université de Lille 1 / *Rapporteur*

Stéphane MANCINI
Professeur à l'ENSIMAG, Institut Polytechnique de Grenoble / *Rapporteur*

Angeliki KRITIKAKOU
Maître de conférence, Universite de Rennes 1 / *Examinatrice*

Biswabandan PANDA
PostDoc à l'INRIA Rennes – Bretagne Atlantique / *Membre*

André SEZNEC
Directeur de recherche à l'INRIA Rennes – Bretagne Atlantique / *Directeur de thèse*

Acknowledgement

First of all, I would like to thank my parents for their unconditional support and love they have showered over all these years. I would also like to equally thank my sister for her constant support and encouragement.

Then, I wish to thank my friends (several of them): each one of them have contributed in his own way. Mentioning their names and their individual contributions would span multiple pages. I would also like to thank my lab and office mates, who were all helpful at different stages of my Ph.D.

I also take this opportunity to thank my teachers during school, college, and university who have taught me subjects and life lessons, which all have kept me in pursuit of higher education. In particular, I sincerely thank Prof. N. Venkateswaran (Waran), my undergraduate adviser who was very instrumental in my pursuing a Ph.D. He gave me an opportunity to work with him during my undergraduate days, which gave exposure to computer architecture.

I thank Professors at UPC, who taught me advanced courses in computer architecture. In particular, I thank Prof. (late) Nacho Navarro, who gave me an opportunity to work with him as an intern at Barcelona Supercomputing Center (BSC). My tenure as an intern exposed me to CUDA (GPU) programming.

I sincerely thank my Ph.D. adviser, Dr. Andre Seznec, for his guidance throughout the course of Ph.D. study. As an adviser, he was very instrumental in driving me along the right direction of work. He was always ready and available for discussions and give feedback on the work. His constant support has resulted in the timely completion of my Ph.D. I sincerely thank you once again!

I thank Dr. Biswabandan Panda for his useful discussions during the course of the second work. In particular, I thank your careful drafting of the paper during its submission. I equally cherish the cricket discussions we have had during the whole of last year!

I also take this opportunity to thank the jury members: Prof. Lionel Lacassagne for presiding over the jury, Prof. Pierre Boulet and Prof. Stephane Mancini for accepting to be the reporters of the thesis. I would also like to thank Prof. Angeliki Kritikakou for accepting to be the examinator of the defense.

Finally, I thank The Almighty for associating me with the right people.

Contents

Résumé en Français

L'écart de performance entre le processeur et la mémoire (DRAM) se creuse de plus en plus, au point d'être appelé "mur mémoire" ". Ce terme se réfère à l'augmentation du nombre de cycles processeur nécessaire pour effectuer un accès mémoire (c'est-à-dire accéder à un circuit en dehors de la puce contenant le processeur) à mesure que de nouvelles générations de processeurs sont conçues. Alors que les architectes étaient au courant de cet écart croissant, Wulf et McKee ont été parmi les premiers chercheurs à formuler ce phénomène dont l'impact allait grandissant. Cet écart de performance croissant entre le processeur et le système mémoire est dû au technologies disparates avec lesquelles les deux composants sont mises en œuvre. En effet, les fabricants de puces (ex., processeurs) sont capables de réduire la taille des transistors, tandis qu'il est difficile de réduire la taille des condensateurs (utilisés dans la mémoire), en raison de problèmes de fiabilité.

Les premières solutionsà ce problème sont axées sur le maintien du processeur occupé quand il est en attente de données venant de la mémoire, en l'autorisantà exécuter des instructions dans le désordre, ainsi que sur l'émission de multiples instructions par cycle, mécanismes qui exploitent le parallélisme niveau instruction (ILP). Le préchargement de données depuis la mémoire dans des mémoires caches directement sur la puce ainsi que l'utilisation de caches multi-niveaux permettent aussi de limite l'impact du " mur mémoire ". Les recherches ultérieures dans cette direction se sont concentrées sur la conception de meilleures techniques micro-architecturales pour améliorer les performances du processeur et cette tendance a continuée au cours d'une décennie entre les années 90 et le début des années 2000. Cependant, des facteurs tels que l'augmentation de la complexité de la conception de tels systèmes, les limites sur l'ILP extractible ainsi que les questions de puissance et de température, ont contraints les fabricants de processeurs de porter leur attention vers la réplication de plusieurs processeurs (cœurs) sur une même puce pour de meilleures performances.

D'une part, la réduction de la taille des transistors décrite par la loi de Moore permet d'intégrer plus de transistors sur une surface donnée. Par conséquent,à chaque génération de processeurs, les fabricants de puces continuent d'intégrer plus de cœurs afin d'utiliser les transistors disponibles. Déjà, il existe des systèmes possédant plusieurs dizaines de cœurs. Cette tendanceà l'intégration de plus de cœurs sur les processeurs est susceptible de se poursuivreà l'avenir, car elle augmente les capacités de calcul en augmentant le nombre d'opérations qui peuvent être effectuées par unité de temps. Dans le même temps, on augmente la quantité de données transférée sur la puce afin

d'être que lesdites données soient traitées. Par conséquent, le système mémoire doit pouvoir fournir des donnéesà tous les cœurs pour une performance soutenue. Cependant, la vitesse du système mémoire reste plus faible que la vitesse du processeur. Cela différence est exacerbée dans les processeurs multi-cœurs car le système mémoire est maintenant partagé par plusieurs cœurs. Typiquement, un accèsà la mémoire passe par différentes structures (files d'attente), qui sont soumisesà divers délais en fonction de la vitesseà laquelle les demandes de mémoire sont traitées ainsi que des différents retards d'ordonnancement. Les délais dusà ces files d'attente impactent les performances des processeurs multi-cœurs.

Ce problème est encore aggravé par la limitation des ressources qui connectent le processeur et la mémoire. En particulier, le processeur est connectéà la mémoire des broches et des canaux métalliques sur la carte mère. Ces broches et fils forment le bus mémoire, qui est coûteux en terme de consommation d'énergie et en latence. L'ITRS prévoit que le nombre de broches qui relient le processeur avec le bus de la mémoire n'augmente que 10 % par an par rapport au nombre de cœurs par processeur, qui double tous les dix-huit mois. Par conséquent, la quantité de données qui peuvent être transféréesà partir du système mémoire vers chaque cœur du processeur est de plus en plus limitée, ce qui ajoute au problème de latence des systèmes mémoires. Au total, le " mur mémoire " peut être considéré comme étant composé du " mur de la latence " et du " mur du débit ".

Problème d'interférences inter-applications Dans les paragraphes précédents, nous avons discuté des limitations technologiques du système mémoire et par conséquent, son manque de performance par rapport aux processeurs modernes. Alors que les contraintes technologiques servent comme un facteur limitant la performance de la mémoire et par conséquent, la performance globale du système multi-cœurs, le caractère partagé de la hiérarchie mémoire,à savoir le cache de dernier niveau et les canaux mémoires liant le processeur au système mémoire, ajouteà ce problème. Les applications ont tendanceà interférer les unes avec les autres au niveau de ces ressources partagées. Par exemple, une ligne de cache peut être expulsée par une ligne de cache d'une autre application, un phénomène connu comme la " pollution de cache ". De même, le grand nombre de requêtes mémoire générées par les applications agressives peuvent retarder le traitement des requêtes générées par d'autres applications. Ces deux problèmes sont exacerbés en présence de préchargement,à cause duquel les interférences dues au partage des ressources mémoires (cache(s) et canaux) pourrait conduireà des ralentissements encore plus prononcés. Par conséquent, les processeurs haute performance utilisent des mécanismes pour gérer les interférences entre les applications au niveau des ressources partagées. L'objectif de cette thèse est d'aborder ces deux problèmes dans le contexte des processeurs many-coeurs, c'est-à-dire les systèmes multi-cœurs avec seize ou plusieurs cœurs sur une puce. Dans les paragraphes qui suivent, nous examinons les deux problèmes dans le contexte des systèmes multi-cœursà grande échelle (many-cœurs).

Gestion du dernier niveau de cache dans le contexte des processeurs many-coeurs

L'augmentation du nombre d'applications en cours d'exécution sur un processeur multi-cœurs augmente la diversité des différentes charges de travail auxquelles le système mémoire doit répondre. En particulier, le cache de dernier niveau est partagé

par des applications se comportant différemment dans leurs accès au système mémoire (i.e., nombre d'accès, motifs d'adresses). Pour utiliser efficacement la capacité de la mémoire cache, l'algorithme de gestion de cache doit prendre en compte les différentes caractéristiques des applications, et donc prioriser l'accèsà la mémoire cache partagée. La nécessité d'attribuer des priorités différentes aux multiples applications s'exécutant de façon concurrente est aussi une conséquence des besoins en performance et en équité des systèmes haute performance commerciaux (i .e., cloud, datacenters). De tels systèmes, où les ressources de la hiérarchie de mémoire sont partagées entre les applications, nécessitent l'attribution de différentes priorités aux applications selon leur caractère critique (pour une certaine définition de critique). Pour répondreà une telle exigence, un nouveau mécanisme pouvant capturer efficacement les différents comportements des applications lors de l'exécution et permettantà la politique de gestion du cache d'appliquer des priorités différentes entre les applications est nécessaire.

Des mécanismes antérieurs ont proposé différentes façons de prédire le comportement des applications relatifsà la réutilisation de lignes de mémoire cache. DIP, TA-DIP et TA-DRRIP peuvent prédire le comportement des applications en mettant en œuvre des politiques complémentaires pour déterminer si une application tire des bénéfices de la présence du cache, ou non. Le prédicteur Hit-Predictor, basé sur les signatures, utilise une région de la mémoire ou de compteurs de programmes (PC) pour prédire le comportement de réutilisation de ces régions de mémoire ou PCs. De même, Evicted Adress Filter (EAF) prédit au niveau de la ligne de cache. Dans tous ces mécanismes, le principe sous-jacent est d'observer les accès réussis ou manqués de groupe de lignes ou de lignes de cache et de prédire le comportement de réutilisation. Ces mécanismes ont été développés pour des systèmes simples, puis étendu aux systèmes multi-cœurs (jusqu'à 4 cœurs). Cependant, une telle approche n'est pas fiable dans le contexte des processeurs many-cœurs. Un défaut de cache ne reflète pas nécessairement le comportement réel d'une application. En effet, les défauts de cache ne sont pas seulement une conséquence du comportement de l'application ; ils sont également une conséquence du comportement des autres applications s'exécutant de façon concurrente. Par conséquent, un nouveau mécanisme qui capture efficacement la façon dont une application peut utiliser le cache est nécessaire. La première partie de cette thèse se concentre sur ce mécanisme.

0.1 Interférence causée par le préchargement

Le préchargement matériel est un mécanisme visantà cacher la latence du système mémoire utilisé dans les processeurs commerciaux. Il fonctionne par l'apprentissage d'un motif dans les accèsà la mémoire cache (il considère les adresses de blocs) et l'envoi de requêtes vers la mémoire afin de précharger des blocs de cache l'application peut accéderà l'avenir. Un préchargement permettant d'obtenir un bloc en temps voulu permet de complètement masquer la latence de la mémoire. Cependant, une ligne de cache préchargée qui n'est au final jamais accédée par le processeur gaspille de la bande passante et peut expulser un bloc de cache utile lors de son insertion. Le préchargement

peut également retarder les accès d'autres applications. Alors que les effets positifs et négatifs du préchargement ont été étudiées dans le contexte des uniprocesseurs (un seul cœurs), son impact est d'autant plus important dans le contexte des systèmes multi-cœurs et many-coeurs. Par conséquent, le préchargement agit comme une épéeà double tranchant et exige une gestion attentive. Cette gestion peut être approchée selon deux dimensions. Tout d'abord, le contrôle du nombre de demandes de préchargement qui sont émises dans le système et d'autre part, la gestion des priorités des demandes de préchargement au niveau cache partagé. Les deuxième et troisième contributions de cette thèse sont axées sur ces deux problèmes.

Des travaux antérieurs sur le contrôle de l'agressivité de l'unité de préchargement tels que hiérarchique (ACVL) et CAFÉINE n'atténuent pas complètement les interférences causées par l'unité de préchargement dans les systèmes multi-cœurs. ACVL utilise plusieurs paramètres, qui selon leurs valeurs, permettent d'inférer qu'une application donnée interfère. Cependant, les différents seuils d'inférence conduisentà des décisions incorrectes sur les applications qui provoquent des interférences car la valeur de seuil d'une métrique donnée ne reflète pas bien le comportement d'une application, en raison du grand nombre d'applications s'exécutant de façon concurrente. De même, CAFÉINE utilise une méthode approximative pour estimer la latence d'accès moyenneà la mémoire pour mesurer les économies en cycles en raison de préchargement. Cependant, cette approximation conduità des décisions biaisées en faveur de préchargement agressif, en ne tenant pas comptes des interférences potentielles. Au total, ces mécanismes fournissent encore de la place pour une meilleure gestion des ressources partagées en présence de préchargement.

Pour traiter les demandes de préchargement dans les caches partagés, les travaux antérieurs tels que Prefetch-Aware Cache Management (PACMAN) et (ICP) observent le comportement de la réutilisation des lignes de cache préchargées dans le contexte de caches de petites tailles. Plus précisément, les deux contributions supposent implicitement que les blocs de cache préchargés sontà usage unique. Cependant, dans le contexte de caches plus grands, cette hypothèse n'est pas toujours vraie ; les lignes de cache préchargées ont un comportement de réutilisation différent. Par conséquent, il y a possibilité de mieux gérer les demandes de préchargement dans les caches partagés.

0.2 Questions de recherche

Dans le sillage des discussions présentées ci-dessus, cette thèse se concentre sur les réponses aux questions de recherche suivantes :

Question 1: Y at-il un mécanisme de gestion de la mémoire cache qui est capable d'isoler efficacement le comportement applications dynamiquement, et qui permet une meilleure gestion de la mémoire cache partagée lorsqu'un grand nombre d'applications partagent le cache?
Question 2: Y at-il un mécanisme pratique qui traite efficacement les interférences causées par l'unité de préchargement dans les systèmes multi-cœurs?
Question 3: Y at-il un mécanisme de gestion de cache qui prend également en compte

les caractéristiques de l'utilisation des données préchargées pour une meilleure gestion?

0.3 Contributions de cette thèse

Dans cette thèse, nous proposons des solutions qui tentent de résoudre chacune des questions de recherche mentionnés ci-dessus. Nos solutions aux problèmes sont les contributions de cette thèse:

0.3.1 Priorisation adaptive et discrète des applications

Pour faire face aux interférences au niveau de la mémoire cache partagée, nous introduisons le numéro d'empreinte afin d'approcher dynamiquement l'empreinte de cache (le nombre de lignes du cache appartenantà l'application) des applicationsà l'exécution. Pour un cache donné, le numéro d'empreinte est défini comme étant le nombre d'accès uniques (adresses de blocs de mémoire cache) qu'une application génère vers ce cache dans un intervalle de temps. Puisque le numéro d'empreinte se rapproche explicitement et quantifie la taille du jeu de travail des applications, il prévoit la possibilité d'attribuer des priorités différentesà chaque application. Puisque nous utilisons des circuits indépendants en dehors la structure de cache principale pour estimer le numéro d'empreinte, il n'est pas affecté par les succès/échecs qui se produisent au niveau du cache partagée. Le comportement d'une application en isolation est donc correctement capturé. Sur la base des valeurs estimées de numéros d'empreintes, l'algorithme de remplacement détermine les priorités à attribuer aux applications. Au total, l'estimation du numéro d'empreinte permet (i) de faire appliquer différentes priorités (discrètes) aux différentes applications et (Ii) capturer efficacement l'utilitaire de cache des applications.

0.3.2 Préchargement passe-bande : Un mécanisme de gestion du préchargement basé sur la fraction de préchargement

Notre solution repose sur deux observations fondamentales sur l'utilité du préchargement et les interférences causées par l'unité de préchargement. En particulier, nous observons (i) une forte corrélation positive existe entre la quantité de demandes de préchargement qu'une application génère et la précision du préchargement. Autrement dit, si la fraction de demandes de préchargement générée est faible, la précision de l'unité de préchargement est également faible. La deuxième observation est que plus le nombre total de demandes de préchargement dans le système est grand, plus la pénalité dueà l'absence d'une ligne dans le cache de dernier niveau est grande. Sur la base de ces deux observations, nous introduisons le concept de fraction de préchargement, afin de déduireà la fois l'utilité du préchargement et les interférences causées par l'unité de préchargement. Notre mécanisme contrôle le flux de demandes de préchargement en fonction de la fraction de préchargement, de façon analogue au filtre passe-bande venant du domaine du traitement du signal. Par conséquent, nous nous référonsà ce mécanisme comme le préchargement passe-bande.

0.3.3 Gestion des demandes de préchargement au cache de dernier niveau partagé en tenant en compte de la réutilisation des lignes

Notre solution permet l'observation que les blocs de cache préchargés ont différentes caractéristiques d'utilisation. Dans une application, certains blocs de cache préchargés sont accédésà plusieurs reprises tandis que d'autres ne sont pas utilisés du tout. Nous observons également que l'utilisation du préchargement varie selon les applications. Par conséquent, nous mesurons les caractéristiques d'utilisation des blocs de cache préchargésà l'exécution pour déterminer leurs priorités lors de remplacements dans le cache de dernier niveau.

0.4 Organisation de la thèse

Le reste de la thèse est organisé comme suit: dans le chapitre 2, nous présentons la gestion de la mémoire cache partagée et la gestion des interférences lors des accèsà la mémoire partagée. Ensuite, nous discutons des mécanismes état de l'art qui ont été proposées pour traiter les deux problèmes que la thèse tente de résoudre. Parallèlement, nous décrivons également leurs lacunes. Dans le chapitre 3, nous présentons notre première solution: un mécanisme adaptatif de priorisation d'application qui gère le cache partagé de dernier niveau. Nous décrivons notre mécanisme de suivi qui capture le numéro d'empreinte des applications, et notre algorithme de remplacement, qui utilise des numéros d'empreintes pour attribuer des priorités aux lignes de cache des différentes applications. Ensuite, nous évaluons notre solution et comparons nos mécanismesà l'état de l'art. Dans le chapitre 4, nous présentons le préchargement passe-bande, et discutons en détail les deux observations qui conduisentà la solution proposée, notre approche pratique pour mesurer la durée moyenne d'une requête mémoire. Nous évaluons ensuite notre contribution et le comparonsà l'état de l'art des mécanismes de contrôle de l'agressivité de l'unité de préchargement. Le chapitre 5 présente notre troisième contribution. Nous décrivons notre mécanisme pour estimer les caractéristiques d'utilisation de l'unité de préchargement puis décrivons l'algorithme qui exploite les caractéristiques d'utilisation pour attribuer des priorités aux demandes de préchargement. Par la suite, nous présentons notre évaluation et la comparaison par rapport aux mécanismes pertinents de l'état de l'art. Enfin, nous présentons notre conclusion et les directions pour la recherche future dans le dernier chapitre.

Chapter 1

Introduction

The growing performance gap between the processor and the DRAM based memory systems is referred to as the Memory Wall. In particular, it refers to the growing increase in number of processor cycles it takes for an off-chip memory access to be serviced by the memory system, as we move from one generation to the next generation of processor chips. While architects had been aware of this growing discrepancy, Wulf and Mckee [WM95] were one of the first researchers to formulate this then impending phenomenon. This growing performance gap between the processor and the memory system is due to disparate technologies with which the two components are implemented, because chip makers are able to scale down the size of transistors, while capacitors do not scale down as gracefully as transistors due to reliability issues of capacitors.

Early solutions to this problem focused on keeping the processor busy while it is waiting for the data from memory by executing instructions out-of-order, mechanisms that exploit Instruction Level Parallelism (ILP) through issuing multiple instructions, pre-fetching data from the memory and storing them on the on-chip caches, and employing multi-level caches. Subsequent research along this direction focussed on designing better micro-architectural techniques to improve processor performance and this trend continued over a decade between the 90s and early 2000s. However, factors such as increasing design complexity of such systems, limits on the extractable ILP, power and thermal issues, forced chip makers to shift their focus toward replicating or employing multiple processor cores on the same chip, Chip Multi-Processing (CMP), or Multi-core processors for higher performance.

On the one hand, transistor scaling driven by Moore's law allows to pack more transistors in a given area of a chip. Consequently, with every generation of processors, chip makers keep packing more cores. Already, there are systems that host multiple processor cores and that are capable of running tens of threads. This trend of packing more processor cores on a chip is likely to continue in the future. Integrating more cores on a processor chip increases the computational capabilities in terms of the number of operations that can be performed per unit of time. At the same time, it increases the amount of data that has to be brought on-chip to process. Therefore, the memory system is expected to provide data to the processor at a higher rate for sustained

performance. However, the speed of the memory system continues to lag behind the processor speed. This becomes even more critical in multi-core processors, because the memory system is now shared by multiple cores. Typically, an access to the memory goes through different queuing structures, which are subject to various queuing delays depending on the rate at which the memory requests are cleared and scheduling delays. The delay on these queuing structures impact the performance of these processor cores.

This problem is further exacerbated by the limitation of resources that connect processor and memory systems. A processor chip is connected to the memory chip by pins and metal channels on the motherboard. These pins and wires form the memory bus, which is expensive in terms of power consumption and implementation [Onu, RKB+09, ITR]. ITRS [ITR] predicts that the number of pins, which connect the processor with the memory bus grows only 10% per year as compared to the number of processor cores, which doubles every eighteen months. Therefore, the amount of data that could be transferred from the memory system to the processor is also limited, which adds to the memory systems' latency problem. While latency wall is prevalent in single-core performance, its combination with limited bandwidth availability limits the performance of a multi-core system. The phenomenon in which the bandwidth becomes a major bottleneck factor in achievable system performance is referred to as the Bandwidth wall [RKB+09]. Altogether, the memory wall can be viewed as to comprise of the latency wall and the bandwidth wall [Pat04].

1.1 Problem of Inter-application Interference

In the previous section, we discussed the technological limitations of the memory system and hence, its performance lag with respect to the processor cores. While technological constraints serve as one limiting factor to memory performance and hence, overall multi-core system performance, the shared nature of the memory-hierarchy, namely the last level cache and off-chip memory channels, adds to this problem. Applications tend to interfere with each other at these shared resources. For example, a cache line of one application could be evicted by a cache line of another application, which is referred to as Cache Pollution. Similarly, memory requests of memory-intensive applications could delay the service of other meek applications, or delay each other at the off-chip memory access. These two problems are exacerbated in the presence of prefetching[1], and interference at these shared resources could lead to severe slow-down [EMLP09, ELMP11, WJM+11, SYX+15, PB15, JBB+15, Pan16, LMNP08, LS11, BIM08]. Therefore, high-performance processors employ mechanisms to manage interference among applications at these shared resources. The goal of this thesis is to address these two problems in the context of large scale multi-core processors. That is, multi-core systems with sixteen or more cores on a chip. In the following subsections, we discuss the two problems in the context of large-scale multi-core systems.

[1] We define prefetching shortly.

1.1.1 Managing last level cache in the context of large scale multi-core systems

Increasing the number of applications that run on a multi-core processor increases the diversity of characteristics and the memory demands the system must cater to. In particular, the last level cache is shared by applications with diverse memory behaviors. For efficiently utilizing the cache capacity, the cache management algorithm must take into account the diverse characteristics of applications, and accordingly prioritize them at the shared cache. Further, the need for enabling different priorities across applications is fueled by the fairness and performance objectives of commercial grid systems. Such commercial systems, where the memory-hierarchy resources are shared among applications, require enforcing different priorities across applications depending on their criticality, cost-involved, and other factors. In order to meet such a requirement, a new mechanism that effectively captures the diverse behaviors of applications at run-time and allows the cache management policy to enforce different priorities across applications, is needed.

Prior mechanisms have proposed different ways to predict the reuse behavior of applications. Dynamic Insertion Policies [QJP+07], Thread Aware-Dynamic Insertion Policies [JHQ+08] and Thread Aware-Dynamic ReReference Interval Prediction [JTSE10] predict the behavior by implementing complementary policies to determine if an application is cache friendly or not. Signature based Hit-Predictor [WJH+11] uses memory region or Program counter (PC) signatures to predict the reuse behavior on these memory or PC regions. Similarly, Evicted Address Filter (EAF) [SMKM12] makes prediction on a per-cache line basis. In all these mechanisms, the underlying principle is to observe the hits or misses on individual cache lines or a group of cache lines and predict the reuse behavior. These mechanisms have been developed for single core systems and then extended to multi-cores (up to 4 cores). However, such an approach is not accurate or reliable in the context of large scale multi-cores. A cache miss does not necessarily reflect the actual reuse behavior of an application, because cache misses are not only a consequence of the reuse behavior of the application; they are also a consequence of the behavior of co-running applications. Therefore, a new mechanism that efficiently captures how well an application could utilize the cache is needed. The first part of the thesis focusses on addressing this problem.

1.1.2 Handling Prefetcher-caused Interference

Hardware prefetching is a memory latency hiding mechanism employed in present day commercial processors. It works by learning a pattern in cache accesses (in block addresses) and issues prefetch requests on the possible cache block addresses that the application may access in the future. A timely prefetch request completely hides the off-chip latency of the memory access. However, a prefetched cache line that is never accessed wastes bandwidth and may evict a useful cache block. Prefetching may also delay the demand requests of other applications apart from wasting bandwidth and polluting useful cache lines. While both positive and negative effects of prefetching have been

studied in single-core context [SMKP07, WJM+11, SYX+15], its impact is more so in the context of multi-core system. Therefore, prefetching acts as a double-edged sword and requires careful management at the shared memory-hierarchy. Handling prefetch requests at the shared memory resources (last level cache and off-chip memory access) can be viewed in two-dimensions. Firstly, controlling the number of prefetch requests that are issued in the system and secondly, to manage the priorities of prefetch requests at the shared cache. The second and third contributions of this thesis are focussed on these two problems.

Prior works on prefetcher aggressiveness control such as Hierarchical Prefetcher Aggressiveness Control (HPAC) [EMLP09] and CAFFEINE [PB15] do not completely alleviate the problem of prefetcher-caused interference in multi-core systems. HPAC uses multiple metrics, which are driven by their threshold values, to infer interference by a given application. However, threshold based inference of interference lead to incorrect decisions on the applications that cause interference because threshold value of a given metric does not reflect the run-time behavior of an application due to interference caused when large number of applications run on the system. Similarly, CAFFEINE uses an approximate method to estimate average memory access latency to measure savings in cycles due to prefetching (that is, prefetch usefulness). However, approximating average memory access latency estimation leads to biased decisions in favor of aggressive prefetching, overlooking interference. Altogether, these mechanisms still provide room for better managing shared resources in the presence of prefetching.

To handle prefetch requests at the shared caches, prior works such as Prefetch-Aware Cache Management (PACMAN) [WJM+11] and Informed Cache Prefetching (ICP) [SYX+15] observe the reuse behavior of prefetched cache lines from the context of small caches. Specifically, the two works implicitly assume prefetched cache blocks as single use (that is, used only once) cache blocks. However, in the context of larger caches, this observation does not always hold true; prefetched cache lines have different reuse behavior. Therefore, there is more opportunity to better manage the prefetch requests at the shared last level caches.

1.2 Research Questions

In the wake of the discussions presented above, this thesis focusses on answering the following research questions:

- Question 1: Is there a cache management mechanism that is able to efficiently isolate applications' run-time behavior, and that allows for better shared cache management when large number of applications share the cache?

- Question 2: Is there a practical mechanism that efficiently addresses prefetcher-caused interference in multi-core systems?

- Question 3: Is there a cache management mechanism that also takes into account prefetched data's reuse characteristics for better management?

1.3 Thesis Contributions

In this thesis, we propose solutions that attempt to solve each of the aforementioned mentioned research questions. Our solutions to the problems are the contributions of this thesis:

1.3.1 Adaptive and Discrete Application Prioritization for Managing Last Level Caches on Large Multicores

To address interference at the shared cache, we introduce the Footprint-number metric to dynamically approximate the last level cache footprint of applications at run-time. Footprint-number is defined as the number of unique accesses (cache block addresses) that an application generates to a cache set in an interval of time. Since Footprint-number explicitly approximates and quantifies the working-set size of applications, it provides scope for implementing different priorities across applications. Since we use independent circuits outside the main cache structure to estimate Footprint-number, it is not affected by the hits/misses that happen at the shared cache. An application's isolated cache utility is reliably captured. Based on the estimated Footprint-number values, the replacement algorithm decides the priorities across applications. Altogether, Footprint-number estimation allows (i) to enforce different (discrete) priorities across applications and (ii) efficiently capture the cache utility of applications.

1.3.2 Band-pass Prefetching : A Prefetch-fraction driven Mechanism for Prefetch Management

Our solution is built upon two fundamental observations on prefetch usefulness and prefetcher-caused interference. In particular, our work finds (i) a strong positive correlation to exist between the amount of prefetch requests an application generates and prefetch-accuracy. That is, if the fraction of prefetch requests generated is low, the accuracy of the prefetcher is also low. The second observation is that, the more the aggregate number of prefetch requests in the system, the higher the miss penalty on the demand misses at the last level cache. Based on the two observations, we introduce the concept of prefetch-fraction to infer both prefetch-usefulness and prefetcher-caused interference. Prefetch-fraction of an application is defined as the fraction of L2 prefetch requests the prefetcher of an application generates with respect to its total requests (demand misses, L1 and L2 prefetch requests). Using prefetch-fraction, our mechanism controls the flow of prefetch requests between a range of prefetch-to-demand ratios, which is analogous to Band-pass filter from signal processing domain. Hence, we refer to this mechanism as Band-pass prefetching.

1.3.3 Reuse-aware Prefetch Request Management : Handling prefetch requests at the shared last level cache

Our solution makes observation that prefetched cache blocks have different use characteristics. Within an application, some prefetched cache blocks are accessed multiple

times while some others are not used at all. We also observe that prefetch use characteristics vary across applications. Therefore, we measure the use characteristics of prefetched cache blocks at run-time to determine their priorities during cache replacements at the last level cache.

1.4 Thesis Organization

The remainder of the thesis is organized as follows: in Chapter 2, we present the background on shared cache management and managing interference at the shared off-chip memory access. Then, we discuss state-of-the art mechanisms that have been proposed to handle the two problems that the thesis attempts to solve. Alongside, we also describe their shortcomings. In Chapter 3, we present our first solution: An Adaptive and Discrete Application Prioritization mechanism that manages shared last level caches. We describe our monitoring mechanism that captures Footprint-numbers of applications, and our replacement algorithm, which uses Footprint-number values to assign priorities to cache lines of applications. Then, we evaluate our solution and compare against state-of-the-art mechanisms. In Chapter 4, we present Band-pass Prefetching, discuss in detail the two observations that lead to our proposed solution, our practical approach to measure average miss service times. We then evaluate our contribution and compare against state-of-the-art prefetcher aggressiveness control mechanisms. Chapter 5 presents our third contribution. We describe our mechanism to estimate prefetch use characteristics and then describe the algorithm that leverages the use characteristics to assign priorities for prefetch requests. Subsequently, we present our evaluation and comparison against the relevant state-of-the-art mechanisms. Finally, we present our conclusion and direction for future research in the last chapter.

Chapter 2

Background

In this chapter, we provide background on shared cache management mechanisms and discuss the state-of-the-art in shared cache management. Then, we provide background on hardware prefetching followed by discussions on state-of-the-art managing prefetching in the context of multi-core processors.

2.1 Cache Management Policies

Shared cache management can be broadly classified into explicit or hard partitioning and implicit or soft partitioning mechanisms. Before going into specific policies under each category, we will first describe these mechanisms.
Hard Partitioning: Hard partitioning mechanisms allocate fixed budget of cache space per application such that the sum of the partitions equals the total cache size. Figure 2.1 shows an example how a cache that is managed by hard partitioning looks. The left part of the figure shows the unmanaged or unpartitioned cache. The right side shows the partitioned cache. This example shows a scenario where an 8-way associative cache is shared by four threads or applications. The cache partitioning algorithm allocates a fixed number of ways per set for every application or thread based on a cache utility model. The model estimates how well a thread makes use of the cache ways allocated to it in terms of reduction in number of misses. When an application finds a miss, the replacement policy selects a cache line within the application's partition of cache ways. If thread A in the figure finds a miss on Set 1, the replacement policy evicts the zeroth way, which is occupied by thread A for replacement.

Soft Partitioning: Soft partitioning mechanism is simply called cache replacement, because there is no explicit allocation of cache ways among applications, and the cache replacement policy just decides to evict the cache line that has the least-priority in the recency order enforced by the policy. If an application finds a miss, the replacement policy can select any cache line (that meets least-priority criterion) irrespective of which application it belongs to. Cache replacement (or, soft partitioning) based cache management is the preferred way since cache partitioning method is constrained by scalability. That is, caches need to be highly associative for cache partitioning to be applied when

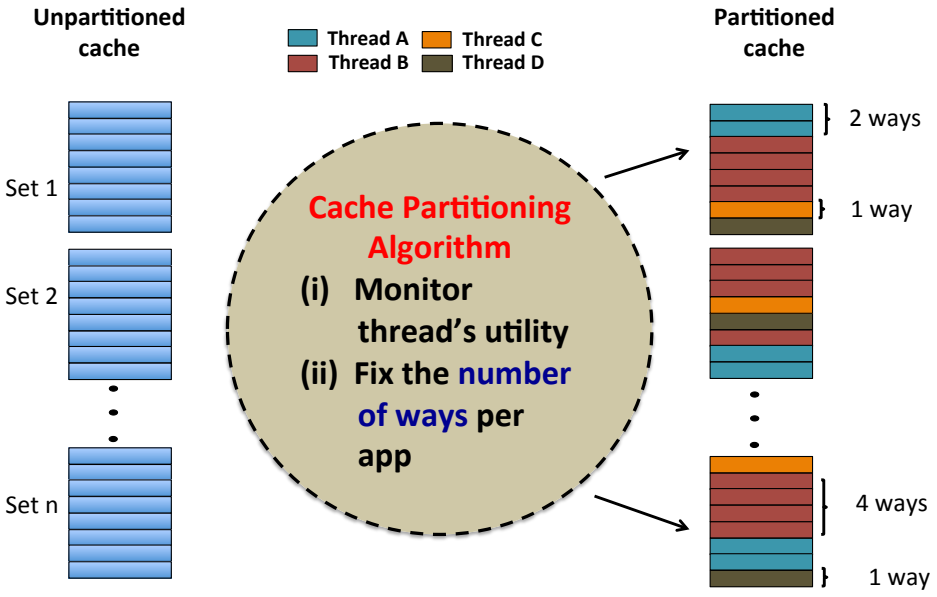


Figure 2.1: An example showing how hard partitioned cache looks

the cache is shared by larger number of applications. We discuss this in detail in the later part of this chapter.

Numerous studies have been proposed in the past under each category. First we discuss implicit cache partitioning schemes and then, discuss cache partitioning schemes.

2.1.1 Cache Replacement Policies

Cache replacement policies base their replacement decisions using reuse distance information. Reuse distance is defined as the number of interleaving cache accesses between consecutive accesses to the same cache line. A cache line with a shorter reuse distance implies that the given cache line will be reused immediately, while a larger reuse distance means that the given cache line will be reused after long period of time. While some approaches explicitly measure the reuse distance of cache lines, most approaches make qualitative prediction on reuse distance values. Because, measuring reuse distance values for individual cache lines require huge bookkeeping with respect to cache block addresses. On the other hand, making qualitative prediction on reuse distance is much simpler in terms of hardware overhead: few bits (2 or 4-bit) information per cache line.

The typical example of reuse distance prediction based cache replacement is the state-of-the-art Least Recently Used (LRU) policy. It works by ordering the cache lines of a given set in a recency order. On a cache miss, the cache line with least priority in the recency order is evicted. For the missing cache line, LRU policy implicitly assigns

Most Recently Used, that is the highest priority in the recency chain. Dynamic Insertion Policy (DIP) [QJP+07] is one of the foremost studies to break this implicit assumption on cache insertions. They break cache replacement into two sub-problems of eviction and insertion. Several works have been proposed in the past that enhance the eviction and insertion decisions of the LRU policy. In the following sub-sections, we discuss cache replacement policies that take different approaches to make insertion and eviction decisions.

2.1.1.1 Insertion Priority Prediction

As mentioned above, cache replacement involves eviction a cache line and inserting the missing cache line. Several mechanisms have fallen in-line with this principle, and in particular with making prediction on cache insertions. Here, we discuss mechanisms that use different approaches to base prediction on insertion priorities of cache lines.

Querishi et.al. observe that the LRU policy thrashes for applications with working set size larger than the cache. Typically, the LRU policy evicts the least recently used cache line and inserts the missing cache line with the most recently used priority. In doing so, LRU policy allocates every missing cache line with highest priority and therefore, allows each of them to stay in the cache for a long duration of time before getting evicted. However, for such applications, DIP observes that thrashing can be avoided by allowing only a part or fraction of the working-set to be allowed to stay longer in the cache, while leaving the remaining or larger part of the working-set to stay much shorter in the cache. To achieve this, DIP alters the insertion policy by inserting the missing cache line with the highest MRU priority only probabilistically (1/32 times). In all other cases, the priority of the cache lines remain at LRU. The idea is that those LRU inserted cache lines are eventually filtered out on subsequent misses and avoiding thrashing. This variant policy is referred to as Bi-modal insertion policy (BIP). However, for recency-friendly applications, LRU is still the best policy. They propose set-dueling to dynamically learn the best policy for a given application. TADIP [JHQ+08] proposes a thread-aware methodology to dynamically learn the insertion priorities in a shared cache environment.

DRRIP [JTSE10] proposes to predict the reuse behavior of cache lines as re-reference interval buckets. That is, cache lines of a set are classified into different re-reference intervals as opposed to a particular re-reference or reuse value. While LRU based recency chain makes relative ordering between cache blocks within a set, RRIP allows multiple cache blocks to exist under same re-reference interval. RRIP uses 2-bit per cache line to encode re-reference interval information which allows a cache block to exist in one among four states or re-reference buckets. The highest state represents immediate re-reference, while the lowest state represents distant re-reference interval. On a cache miss, the cache line with distant re-reference interval value (3) is chosen for eviction. Unlike LRU or DIP, RRIP does not insert a cache line with highest priority, immediate re-use. From studies performed on applications from gaming and multimedia segments, RRIP discovers new access patterns namely, scan, a long sequence of cache lines with no reuse and mixed pattern where recency-friendly pattern mixed with scan

access pattern. To handle workloads with different access patterns, RRIP propose Static Re-reference Interval Prediction, SRRIP and Bi-modal Re-reference Interval Prediction, BRRIP. While SRRIP handles mixed and scan type of access patterns, BRRIP handles thrashing patterns. SRRIP policy inserts the cache lines with RRIP value 2, which is neither immediate or distant re-reference interval, and called intermediate re-reference interval. Doing so, allows cache lines of recency-friendly pattern to stay long enough until they are reused. At the same time, cache lines of scan type applications are evicted soon as compared to inserting with highest, immediate priority. Insertion value of 2 is experimentally determined.

As with DIP, RRIP uses set-dueling to dynamically learn the best of the two polices for an application. t uses the same methodology [JHQ+08] to manage applications in shared cache environment. A thread-aware RRIP (TA-DRRIP) is our baseline cache replacement policy.

While DIP and RRIP policies make insertion priority prediction at the granularity of applications, mechanisms like Signature-based Hit Prediction, SHiP [WJH+11] and Evicted Address Filter, EAF [SMKM12] make predictions at finer granularities. SHiP classifies cache accesses into groups that are identified by PCs, Instruction Sequences or Memory regions. Grouping based on PC signatures yield the best results. To make insertion decisions, SHiP uses a table of 2-bit saturing counters that stores the hit/miss information per signature. It further uses an outcome bit per cache line that indicates a hit or miss on it. When a cache line is evicted, based on the outcome bit the counter associated with that signature is incremented or decremented. When a new cache line is inserted, if value of the counter is maximum, it indicates the cache lines associated with that signature are highly reused. The new cache line is inserted with intermediate (like SRRIP) re-reference prediction. While a minimum value indicates no or poor reuse and the new cache line is inserted with distant re-reference (like BRRIP) prediction.

Evicted Address Filter (EAF) proposed by Seshadri et.al. further enhances the prediction granularity to individual cache lines. A filter is used to decide the insertion priority of the missing cache line. The idea is to hold the evicted cache addresses in the filter, which size is same as that of the cache. If a cache access misses at the cache but hits in the filter is an indication that the cache block is prematurely evicted from the cache. Therefore, the missing cache line is inserted with intermediate (like SRRIP) priority. If the cache access misses both in the cache and filter, it is inserted with distant (like BRRIP) priority.

All these approaches use only binary (SRRIP or BRRIP) insertion policies. Under large scale multi-core context, where applications needs to be differently prioritized, these mechanisms cannot be adapted to enable such discrete prioritization. Further, SHiP and EAF predict priorities at the granularity of individual or regions of cache lines and appear as finer classification mechanisms[1]. However, in commercial designs [CMT][arc], which use a software-hardware co-designed approach to resource management, the system software decides fairness or performance objectives only at an ap-

[1]Finer classification we mention here should not to be confused with discrete classification that we propose. We refer to discrete as having (> 2) priorities across applications.

plication granularity. Hence, it is desirable that the cache management also performs application level performance optimizations.

2.1.1.2 Reuse distance prediction

The mechanisms discussed above make only qualitative estimate (predictions) on the reuse behavior of cache lines and classify them as to have either intermediate or distant reuse. However, some studies [TH04, KPK07, KS08, PKK09, EBSH11, SKP10, DZK+12] explicitly compute the reuse distance value of cache lines at run-time and perform replacements using the explicit value of reuse distance. Since the reuse distances of cache lines can take wider range of values, measuring reuse distance at run-time is typically complex, requires significant storage and modifying the cache tag arrays to store reuse distance values for cache lines. Here, we describe some of the important works in this domain.

[TH04] predicts the inter reference gap of cache lines and assigns them as weights. Their work is based on the observation that the inter reference gap distribution of a cache block takes only few discrete values and that cache blocks with same re-reference counts have same inter reference gap distribution. Thus, cache blocks are grouped (classified) based on the number of re-references. Essentially, classification of blocks signify different inter reference gaps (or, priorities). During its lifetime, a cache block moves from one class to another before being evicted. [KPK07, PKK09] capture the reuse distance of cache blocks using the PCs that access them. They observe only few PCs to contribute to the most of the cache accesses. However, these techniques apply to single-thread context. Since reuse distance computation for all cache blocks incurs significant overhead, some studies have proposed to sample cache accesses and compute the reuse distance for select cache blocks [EBSH11, SKP10, DZK+12].

[SKP10] observe that in multi-threaded environments, the timing of interactions between the threads does not affect stack distance computation. Hence, stack distance can be computed in parallel for all threads. They use sampling to track the reuse distance of individual threads. In their approach, the stack distance of cache lines receiving invalidation (on coherence update) from another thread is approximated to have very distant reuse (maximum reuse distance value). This is because such a cache line would receive a coherence miss anyway. This assumption may not be optimal in certain cases. For example, assume certain threads share a cache block and frequently access/update the cache block. Threads which read the updated block gets an coherence miss and subsequently, gets from the other thread. Forcing a distant reuse on such cache blocks could make the replacement policy to inadvertently give low priority and cause early evictions resulting in frequent off-chip accesses. Further, their approach may be counter-productive when combined with any on-chip cache management technique as in [KKD13].

[DZK+12] propose the protecting distance (PD) metric to protect cache lines until certain number of accesses. Also, they propose a hit-rate model which dynamically checks if inserting a cache line would improve the hit-rate. If not, the cache line is bypassed. They extend the hit-rate model to decide per-thread PD that maximizes

the hit-rate of the shared cache. Computing protecting distance is quite complex and incurs significant hardware overhead in terms of logic and storage. For large number of applications, computing optimal PDs may require searching across a large reuse distance space. Conversely, ADAPT requires only tracking a limited number of accesses (sixteen) per set and simple logic to compute Footprint-number.

[MRG11] propose a novel cache organization that builds on the idea of delinquent PCs. They logically partition the cache as main-ways and deli-ways. While cache lines accessed by all PCs can access main-ways, only cache lines accessed by certain delinquent PCs (policy which is based on the observation that delinquent PCs have different eviction to reuse distance. The idea is to store the lines evicted from the main-ways into deli-ways and retain the cache lines for duration beyond their eviction. There are two problems with their approach. Firstly, for their scheme to work well, the cache needs to have larger associativity, which adds significant energy overhead. Secondly, when there are large number of applications sharing the cache, their algorithm may not be able to find the optimal set of delinquent PCs across all applications and assign deli-ways among them.

2.1.1.3 Eviction priority prediction

Victim selection techniques try to predict cache lines that are either dead or very unlikely to be re-used soon [LR02, LFHB08, LFF01a, WAM13]. A recent proposal, application-aware cache replacement [WAM13] predicts cache lines with very long re-use distance using hit-gap counters. Hit-gap is defined as the number of accesses to a set between two hits to the same cache line. Precisely, the hit-gap gives the maximum duration for which the cache line should stay in the cache. On replacements, a cache line residing closer to/beyond this hit-gap value is evicted. In many-cores, under their approach, certain recency-friendly applications could get hidden behind memory-intensive applications and would suffer more misses. However, ADAPT would be able to classify such applications and retain their cache lines for longer time. Further, this mechanism requires expensive look-up operations and significant modifications to the cache tag array.

2.1.2 Cache Bypassing

Bypassing cache lines was proposed in many studies [CT99, JCMH99, McF92, GAV95, GW10, KKD13]. Run-time Cache Bypassing [JCMH99] proposes to bypass cache lines with low-reuse behaviors while few others try to address conflict misses by bypassing cache lines that could pollute the cache. All these techniques either completely bypass or insert all requests. For thrashing applications, retaining a fraction of the working set is beneficial to the application [QJP+07]. However, in many-core caches, such an approach is not completely beneficial. Inserting cache lines of thrashing applications with least-priority still pollutes the cache. Instead, bypassing most of their cache lines is beneficial both to the thrashing application as well as the overall performance. As we show in the evaluation section, bypassing least-priority cache lines is beneficial to other replacement policies as well.

Segmented-LRU [GW10] proposes probabilistic bypassing of cache lines. The tag of the bypassed cache line and the tag of the victim cache line (which is actually not evicted) are each held in separate registers. If an access to the virtual victim cache line is found to occur ahead of the bypassed cache line, the bypass is evaluated to be useful. This mechanism functions well in single-core context and small-scale multi-cores. However, in many-cores, as demonstrated in the motivation section, observing the hits and misses on the shared cache is not an efficient way to decide on policies as they may lead to incorrect decisions. On the contrary, ADAPT decides to bypass cache lines based on Footprint-number of applications which does suffer from the actual activity of the shared cache since the application behavior is determined explicitly.

[KKD13] studies data locality aware management of Private L1 caches for latency and energy benefits. An on-chip mechanism detects locality (spatial and temporal) of individual cache lines. On an L1 miss, only lines with high locality are allocated at L1 while the cache lines with low locality are not allocated at L1 (just accessed from L2). Thus, caching low locality data only at the shared cache (L2) avoids polluting the private L1 cache and saves energy by avoiding unnecessary data movement within on-chip hierarchy. The principal difference from our approach is that they manage private caches by forcing exclusivity on select data while we manage shared caches by forcing exclusivity on select application cache lines.

2.1.3 Cache partitioning techniques

Cache partitioning techniques [CS07, NLS07, XL09, QP06, GST13] focus on allocating fixed-number of ways per set to competing applications. Typically, a shadow tag structure [QP06] monitors the application's cache utility by using counters to record the number of hits each recency-position in the LRU stack receives. Essentially, the counter value indicates the number of misses saved if that cache way were not allocated to that application. The allocation policy assigns cache ways to applications based on their relative margin of benefit. The shadow tag mechanism exploits the stack property of LRU [MGST70].

While these studies are constrained by the number of cache ways in the last level cache and hence, suffer from scalability with number of cores, some studies have proposed novel approaches to fine-grained cache partitioning [SK11, MRG12, BS13] that breaks the partitioning-associativity barrier. These mechanism achieve fine-grained (at cache block level) through adjusting the eviction priorities. Jigsaw [SK11] shares the same hardware mechanism as Vantage [BS13], but uses a novel software cache allocation policy. The policy is based on the insight that miss-curves are typically non-convex and the convex-hull of the miss-curves provides scope for efficient and a faster allocation algorithm. Vantage, however, uses the same lookahead allocation policy ($O(N^2)$ algorithm) as in UCP. PriSM [MRG12] proposes a pool of allocation policies which are based on the miss-rates and cache occupancies of individual applications.

Essentially, these mechanisms require quite larger associative shared cache. For tracking per-application utility, 256-way associative, LRU managed shadow tags are required [SK11][BS13]. Further, these techniques require significant modification to the

existing cache replacement to adapt to their needs. On the contrary, ADAPT is simple and does not require modification to the cache states. Only the insertion priorities are altered.

2.2 Cache Management in the presence of Prefetching

Hardware prefetching helps to hide long latency memory operation by predicting the future access of an application. When the prefetch request arrives before the demand access (for which prefetch was sent) needs the data, it becomes timely and useful. While a prefetch request that is in transit (yet to arrive) from the main memory may still be useful depending on the criticality of the missing data. A prefetched data that is never used or accessed by the demand stream wastes the memory bandwidth and could potentially cause pollution of useful data at the shared cache. The opportunity cost of wasted bandwidth and pollution becomes higher in the context of multi-core systems, where memory bandwidth and last level cache is shared. Therefore, prefetch requests have to be managed at both the cache as well as off-chip memory access.

2.2.1 Managing prefetch requests at the shared cache

Several studies have proposed prefetch request handling at the caches [DDS95, BC95, Lin01, LRBP01, LFF01b, ZL03, CFKL95, PGG+95, AB05, STS08, SMKP07, WJM+11, SYX+15]. In this following, we only discuss prefetch-aware cache management mechanisms in the context of shared last level caches.

Wu et. al. [WJM+11] find that treating both prefetch and demand requests on cache insertions and cache update operations is detrimental to performance. Because, the use characteristics of demands and prefetch requests are different. At the last level cache prefetch requests are either never accessed or mostly used only once. That is, after the first access by demand, the prefetched data is never accessed again as subsequent accesses is filtered by the level two cache. Therefore, in order to avoid such prefetched cache block occupying cache space for longer duration of time and polluting, they propose to modify the promotion and insertion policies for prefetch requests. On prefetch-hits, the cache line is never promoted, while on cache insertion, prefetched cache lines are always inserted with distant reuse priority. Altogether, they propose Prefetch-Aware Cache MANagement (PACMAN) for handling prefetch requests at the LLC.

Seshadri et. al. propose Informed Cache Prefetching, ICP to further enhances cache management in the presence of prefetching by accounting prefetch-accuracy information. When prefetched cache blocks are inserted with distant priority, chances are high that a useful prefetched block could get evicted before a demand actually uses it. Inserting a prefetched cache block with high priority avoids this problem. However, since prefetched cache blocks are mostly used only once, inserting them at MRU position would result in pollution. Therefore, they propose two mechanisms. The first mechanism, called ICP-Demotion, demotes the cache line to LRU priority on prefetch-hits. The second mechanism called ICP-AP uses a prefetch-accuracy predictor to determine the insertion

priority. If the predictor predicts the prefetcher to be accurate, the mechanism inserts it with MRU priority, otherwise, the cache line is inserted with LRU priority.

2.2.2 Prefetch-filter based Techniques

Several studies have proposed efficient prefetch-filtering mechanisms for stream based prefetchers. Zhuang and Lee [ZL03] propose a filtering mechanism that uses a history table (2-bit counter indexed by Program Counter or cache block address) to decide the effectiveness of prefetching. Prefetch requests are issued depending on the outcome of the counter. Lin et. al. [LRBP01] propose to use filter to reduce the number of useless prefetch requests issued by the prefetcher. The basic idea is to store a bit-vector for each region of data that is tracked. An access to a block sets the bit. They introduce several metrics like population count PC to count the number bits set, longest run length, LRL to count the number of contiguous blocks accessed as measures of locality captured by bit-vectors, and few correlation metrics that take into account the history of the bit-vector in the previous epochs (epoch : time between access to the same cache block within a tracked region). Among the local (of the same region), spatial (of two adjacent regions), and global (of any two regions) correlations, local correlation gives the best results. On a miss, the local correlation function $fracPC(AB)log2(P)$, where A and B represent the bit-vector value in the current and previous epochs, while 'P' represents the number of bits in the density vector. Mechanisms proposed by Kumar et. al. [KW98] and Somogyi [SWA+06] are similar in principle. However, their approach to learn the prefetchable cache lines vary. Kumar et. al.[KW98] uses smaller memory regions called Sectors, and uses two tables to record the footprint within each sector. Spatial Footprint History Table (SHT) records the history while the Active Sector Table (AST) records the footprint of the active sector. To index into the SHT, bits from the cache line address of the first access to that sector and the instruction that generated the first access are hashed. Similarly, Somogyi et. al. uses three tables namely, Filter Table (FT), Accumulation Table (AT), which together form the Active Generation Table (AGT), and Pattern History Table (PHT). Training on a memory region happens between the first demand access and the first invalidation or eviction of a cache line within the region. Filter table extracts the spatial signature, which is PC+offset, where offset is the offset from the starting of the spatial region. The accumulation table records the accesses in the corresponding bits within the spatial region. Using the Pattern History Table (PHT), which is indexed by the spatial signature, SMS sends prefetch requests.

Hur and Lin propose Adaptive Stream Detection mechanisms [HL06, HL09] for effectively detecting short streams using dynamic histograms. The basic idea behind the two mechanisms is to dynamically construct histogram distribution for different stream lengths (in steps of 1). The distribution gives the probability that a given access is part of a current or a longer stream. To issue a prefetch request, the prefetcher tests if the probability of an access being part of a longer stream as compared to the current stream. If the condition is satisfied, the prefetcher issues a prefetch request. Otherwise, the request is not issued. They further extend this idea to issue multiple

prefetch requests. All these mechanisms are developed for single-core processors. In multi-core systems, these mechanisms cannot capture interference due to prefetching. They must be augmented with techniques to capture interference. On the other hand, our mechanism uses prefetch-fraction to determine both the accuracy (usefulness) of prefetching as well as prefetcher-caused interference.

2.2.3 Adaptive Prefetching Techniques

Jimenez et. al. [JGC+12, JBB+15] and Sinharoy et. al. [SKS+11] propose a software based approach to perform dynamic prefetcher aggressiveness control. Both studies explore the benefit of prefetching at run-time and adapt the prefetcher depending on the prefetch benefit or memory bandwidth saturation model. Our mechanism is hardware based and performs prefetch aggressiveness control based on prefetch-fraction. Similarly, Panda proposes SPAC [Pan16], a mechanism that attempts to minimize search space when looking for prefetch configurations that together satisfy system-wide harmonic-speedup goal. Basically, to gauge the usefulness of prefetching, Panda introduces a proxy metric of Prefetcher-caused Fair-speedup (PFS). Prefetchers are classified into two groups : meek and strong based on the fraction of prefetch requests generated per miss (PPM). Prefetchers that fall below the average of PPM are classified as meek, while prefetchers whose PPM fall above the average are classified as strong. SPAC explores across the toned-down search space (5 configurations) of prefetching in short successive intervals and uses the best for the larger interval. Instead our mechanism simply computes prefetch-fraction and controls only the application that issues most of the prefetch requests, while allowing other prefetcher to be aggressive. ABS [AGIn+12] is a prefetcher aggressiveness control mechanism that is proposed for systems where prefetching is employed at the banks of the shared last level cache. Our mechanism can be applied on top of such systems: prefetcher of the individual banks can be treated as a prefetcher-resource and then monitor the requests from each banks. When there is interference, the prefetcher of the bank that issues the highest prefetch-fraction can be throttled-down.

2.3 Conclusion

In this chapter, we presented the background on state-of-the-art cache management mechanisms. From the description, we understand that prior mechanisms on insertion priority prediction suffer from ineffectiveness in predicting the reuse behavior of applications due to their observance of hit/miss statistics at the shared caches. On the other hand, mechanisms that explicitly predict the reuse distance of cache blocks of application require exorbitant storage-cost to explicitly measure reuse distance values. Similarly, cache partitioning mechanisms require larger associative caches which is energy prohibitive. With state-of-the-art mechanisms suffer from shortcomings, we conclude a new mechanism to manage shared caches in large-scale multi-cores is required.

In the context of managing interference in the presence of prefetching, prior mechanisms have proposed mechanisms to reduce interference by controlling the prefetcher

aggressiveness. However, these mechanisms are ineffective in estimating prefetcher-caused interference in the context of large-scale multi-cores. Mechanisms that handle prefetch requests at the shared cache treat prefetch and demands differently which is inefficient in the context of large-scale multi-core caches. Similarly, mechanisms that handle interference between prefetch and demand requests at the off-chip memory access are ineffective in identifying interference. Altogether, handling interference at the shared memory resources require new mechanisms.

Chapter 3

Discrete and De-prioritized Cache Insertion Policies

In this chapter, we present our first contribution. This contribution was presented at IPDPS 2016, where it received the best paper award in the computer architecture track. We first motivate our work on the need for a new cache management for large scale multi-cores and then, present our proposed mechanism. In the subsequent section, we show our evaluation and discuss the results.

3.1 Introduction

In multi-core processors, the Last Level Cache (LLC) is usually shared by all the cores (threads)[1]. The effect of inter-thread interference due to sharing has been extensively studied in small scale multi-core contexts [QJP+07, JHQ+08, GW10, QP06, NLS07, XL09, JTSE10, WJH+11, Iye04, CGB+12, SMKM12]. However, with advancement in process technology, processors are evolving towards packaging more cores on a chip. Future multi-core processors are still expected to share the last level cache among threads. Consequently, future multi-cores pose two new challenges. Firstly, the shared cache associativity is not expected to increase beyond around sixteen due to energy constraints, though there is an increase in the number of cores sharing the cache in multi-core processors. Hence, we are presented with the scenario of managing shared caches where ($\#cores \geq \#llc\_ways$). Secondly, in large scale multi-core systems, the workload mix typically consists of applications with very diverse memory demands. For efficient cache management, the replacement policy must be aware of such diversity to enforce different priorities across applications. Moreover, in commercial grid systems, the computing resources (in particular, memory-hierarchy) are shared across multiple applications which have different fairness and performance goals. Either the operating system or the hypervisor takes responsibility in accomplishing these goals. Therefore, the hardware must provide scope for the software to enforce different priorities for the applications. Altogether, the cache replacement policy must satisfy two requirements

[1]Without loss of generality, we assume one thread/application per core.

(i) allow enforcing discrete priorities across applications and (ii) efficiently capture an application's run-time behavior.

Problem: Prior studies [QJP+07, JHQ+08, GW10, JTSE10, WJH+11, SMKM12] have proposed novel approaches to predict the reuse behavior of applications and, hence their ability to utilize the cache. The typical approach is to observe the hits/misses it experiences as a consequence of sharing the cache and approximate its behavior. This approach fairly reflects an application's ability to utilize the cache when the number of applications sharing the cache is small (2 or 4). However, we observe that this approach may not necessarily reflect an application's ability to utilize the cache when it is shared by a large number of applications with diverse memory behaviors. Consequently, this approach leads to incorrect decisions and cannot be used to enforce different priorities across applications. We demonstrate this problem with an example.

Solution: Towards this goal, we introduce the metric Footprint-number. Footprint-number is defined as the number of unique accesses (cache block addresses) that an application generates to a cache set in an interval of time. Since Footprint-number explicitly approximates the working set size, and quantifies the application behavior at run-time, it naturally provides scope for discretely (distinct and more than two priorities) prioritizing applications. We propose an insertion-priority-prediction algorithm that uses application's Footprint-number to assign priority to the cache lines of applications during cache replacement (insertion) operations. Since Footprint-number is computed at run-time, dynamic changes in the application behavior are also captured. We further find that probabilistically de-prioritizing certain applications during cache insertions (that is, not inserting the cache lines) provides a scalable solution for efficient cache management.

3.2 Motivation

In this section, we motivate the need for altogether a new mechanism to capture application behavior at run-time and a replacement policy that differently prioritizes applications.

Cache Management in large-scale multi-cores:

A typical approach to approximate an application's behavior is to observe the hits and misses it encounters at the cache. Several prior mechanisms [JHQ+08, JTSE10, GW10, WJH+11, SMKM12] have used this approach: the general goal being to assign cache space (not explicitly but by reuse prediction) to applications that could utilize the cache better. This approach works well when the number of applications sharing the cache is small (2 or 4 cores). However, such an approach becomes suboptimal when the cache is shared by a large number of applications. We explain with set-dueling [QJP+07] as an example.

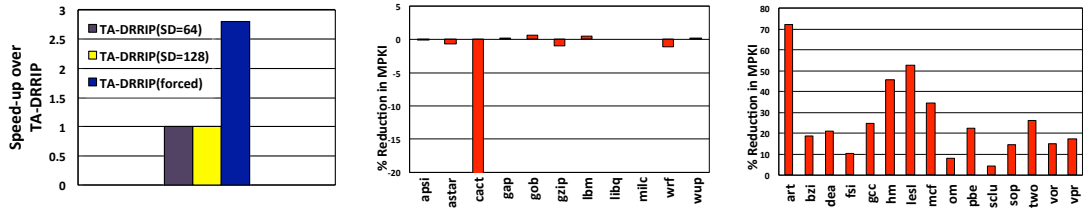
Set-dueling: a randomly chosen pool of sets (Pool A for convenience) exclusively implements one particular insertion policy for the cache lines that miss on these sets. While another pool of sets (Pool B) exclusively implements a different insertion policy. A saturating counter records the misses incurred by either of the policies: misses on Pool A

increment, while the misses on Pool B decrement the saturating counter, which is 10-bit in size. The switching threshold between the two policies is 512. They observe that choosing as few as 32 sets per policy is sufficient. Thread-Aware Dynamic ReReference Interval Predictions, TA-DRRIP[JTSE10] uses set-dueling to learn between SRRIP and BRRIP insertion policies. SRRIP handles scan (long sequence of no reuse cache lines) and mixed (recency-friendly pattern mixing with scan) type of access patterns, BRRIP handles thrashing (larger working-set) patterns.

TA-DRRIP learns SRRIP policy for all classes of applications, including the ones with working-set size larger than the cache. However, applications with working-set size larger than the cache cause thrashing when they share the cache with other (cache-friendly) applications. Based on this observation, by explicitly preventing thrashing applications from competing with the non-thrashing (or, cache friendly) applications for cache space, performance can be improvement. In other words, implementing BRRIP policy for these thrashing applications will be beneficial to the overall performance. Figure 3.1a confirms this premise. The bar labeled TA-DRRIP(forced) is the implementation where we force BRRIP policy on all the thrashing applications. Performance is normalized to TA-DRRIP. From the figure, we observe the latter achieves speed-up close to 2.8 over the default implementation of TA-DRRIP, which records the number of misses caused by the competing policies and making inefficient decisions on application priorities. The experiments are performed on a 16MB, 16-way associative cache, which is shared by all sixteen applications. Table 3.3 shows other simulation parameters. Results in Figure 3.1 are averaged from all the 60 16-core workloads. Also, from bars 1 and 2, we see that the observed behavior of TA-DRRIP is not dependent on the number of sets dedicated for policy learning.

Figures 3.1b and 3.1c show the MPKIs of individual applications when thrashing applications are forced to implement BRRIP insertion policy. For thrashing applications, there is little change in their MPKIs, except cactusADM. cactusADM suffers close to 40% increase in its MPKI and 8% reduction in its IPC while other thrashing applications show a very marginal change in their IPCs. However, non-thrashing applications show much improvement in their MPKIs and IPCs. For example, in Figure 3.1c, art saves up to 72% of its misses (in MPKI) when thrashing applications are forced to implement BRRIP insertion policy. Thus, thrashing applications implementing BRRIP as their insertion policy is beneficial to the overall performance. However, in practice, TA-DRRIP does not implement BRRIP for thrashing applications and loses out on the opportunity for performance improvement. Similarly, SHiP[WJH$^+$11] which learns from the hits and misses of cache lines at the shared cache, suffers from the same problem. Thus, we infer that observing the hit/miss results of cache lines to approximate application behavior is not efficient in the context of large-scale multi-cores.

Complexity of other approaches :
While the techniques that predict the individual application's behavior from its shared cache behavior are not efficient, reuse distance based mechanisms [TH04, KPK07, KS08, PKK09, EBSH11, SKP10, DZK$^+$12] provide a fair approximation of the application's behavior. However, to accurately predict the reuse behavior of individual cache accesses



(a) Forced BRRIP priority (b) MPKI for thrashing applications (c) MPKI reduction for others

Figure 3.1: Impact of implementing BRRIP policy for thrashing applications

of the applications involves significant overhead due to storage and their related bookkeeping operations. Further, these techniques are either dependent on the replacement policy [EBSH11, SKP10] or require modifying the cache tag arrays [KPK07, PKK09, DZK+12]. Similarly, some cache partitioning techniques do not scale with the number of cores while others scale but incur significant overhead due to larger (up to 128/256-way) LRU managed shadow tag structures [SK11, MRG12, GST13, BS13].

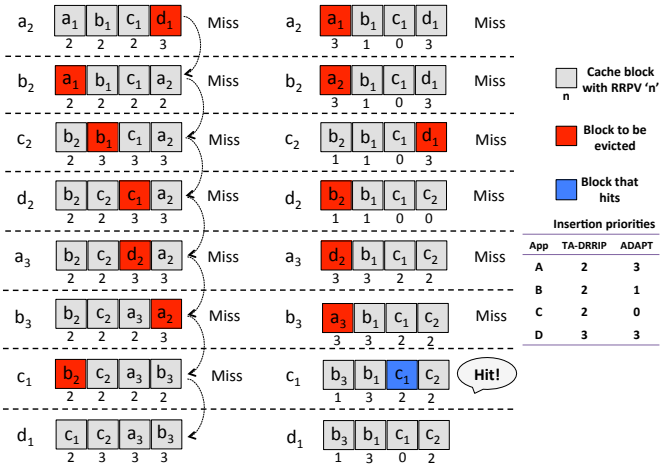
From these discussions, we claim that a simple, efficient and scalable cache monitoring mechanism is required. Further, recall that many-core processors require a cache replacement policy to be application-aware and enforce discrete (> 2) priorities. Therefore, an efficient cache management technique must augment a cache monitoring mechanism that conforms to the above goals as well as allow the cache replacement policy to enforce discrete priorities.

3.2.1 A case for discrete application prioritization:

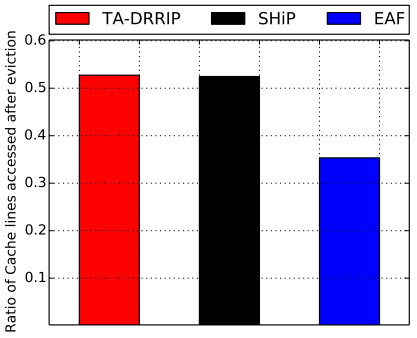
Before presenting our proposed mechanism, we present an example to make a case where discretely prioritizing application is beneficial in the context of large multi-cores where the number of applications or threads sharing in the cache is tight with the number of ways/ associativity of the shared cache.

Like TA-DRRIP, ADAPT also uses 2-bit RRPV counter per cache line to store the predictions and during replacements, it evicts the cache line with RRPV 3. However, the difference is only on insertions, where ADAPT makes (four) discrete predictions. ADAPT implemented in this example does not exactly match the one that will be described in the next section: the example is only to demonstrate the benefit of discrete prioritization.

We assume four applications share a 4-way associative cache. Let $A : \{a1, a2, a3, a4\}^{k1}$, $B : \{b1, b2, b3\}^{k2}$, $C : \{c1, c2\}^{k3}$ and $D : \{d1, d2, d3, d4, d5, d6\}^{k4}$ be the sequences of accesses to cache blocks by applications A, B, C and D, respectively, during a certain interval of time. Let k1=3, k2=1, k3=3 and k4=4 denote the number of reuses to the given access sequence. Assuming a fair scheduling policy, we have the following combined accesses sequence : S1 :{a1,b1,c1,d1}, S2 :{a2,b2,c2,d2}, S3 :{a3,b3,c1,d3}, S4 :{a4,-,c2,d4} etc. Also, let us assume all accesses update their RRPV on hits.



(a) TA-DRRIP vs ADAPT (discrete prioritization)



(b) Ratio of accesses that fall within the first 256 misses in the set after their eviction

Figure 3.2: a) Benefit of discrete prioritization b) Ratio of Early Evictions

Figure 3.2a shows our example. The boxes represent cache tag storing the cache block address and the number below each box represents the block's RRPV. TA-DRRIP inserts cache lines of applications A, B and C with RRPV 2 and cache lines of application D with RRPV 3. On the other hand, ADAPT inserts the cache lines with discrete priorities. Cache lines of applications A and D are inserted with RRPV 3 and, cache lines of application B and C are inserted with RRPV 1 and 0, respectively. Assignment of priorities by ADAPT is the subject of next section. From the figure, we see cache line c1, which is inserted with RRPV 0 is able to survive until its next use. But, TA-DRRIP is not able to preserve cache line c1. c1 is evicted just few accesses before its reuse. From experiments, we observe significant fraction of cache lines to suffer from such evictions. In particular, Figure 3.2b, shows that close 52% of reuses (that miss) in the cache fall within the first 256 misses in the set under TA-DRRIP and SHiP algorithms.

3.3 Adaptive Discrete and de-prioritized Application PrioriTization

Adaptive Discrete and de-prioritized Application PrioriTization, ADAPT, consists of two components: (i) the monitoring mechanism and (ii) the insertion-priority algorithm. The first component monitors the cache accesses (block addresses) of each application and computes its Footprint-number, while the second component infers the insertion priority for the cache lines of an application using its Footprint-number. Firstly, we describe the design, operation and cost of the monitoring mechanism. Then, we describe in detail the insertion-priority algorithm.

3.3.1 Collecting Footprint-number

Definition: Footprint-number of an application is the number of unique accesses (block addresses) that it generates to a cache set. However, during execution, an application may exhibit change in its behavior and hence, we define its Sliding Footprint-number [2] as the number of unique accesses it generates to a set in an interval of time. We define this interval in terms of the number of misses at the shared last level cache since only misses trigger new cache blocks to be allocated. However, sizing of this interval is critical since the combined misses of all the applications at the shared cache could influence their individual (sliding) Footprint-number values. A sufficiently large interval mitigates this effect on Footprint-number values. To fix the interval size, we perform experiments with 0.25M, 0.5M, 1M, 2M and 4M interval sizes. Among, 0.25, 0.5 and 1M misses, 1M gives the best results. And, we do not observe any significant difference in performance between 1M and 4M interval sizes. Further, 1 Million misses on average correspond to 64K misses per application and are roughly four times the total number of blocks in the cache, which is sufficiently large. Hence, we fix the interval size as 1M last level cache misses.

Another point to note is that Footprint-number can only be computed approximately because (i) cache accesses of an application are not uniformly distributed across cache sets, (ii) tracking all cache sets is impractical. However, a prior study [QP06] has shown that the cache behavior of an application can be approximated by sampling a small number of cache sets (as few as 32 sets is enough). We use the same idea of sampling cache sets to approximate Footprint-number. From experiments, we observe that sampling 40 sets are sufficient.

Design and Operation: Figure 3.3a shows the block diagram of a cache implementing ADAPT replacement algorithm. In the figure, the blocks shaded with gray are the additional components required by ADAPT. The test logic checks if the access (block address) belongs to a monitored set and if it is a demand access[3], and then it passes the access to the application sampler. The application sampler samples cache accesses (block addresses) directed to each monitored set. There is a storage structure and a saturating counter associated with each monitored set. The storage structure is essentially an array which operates like a typical tag-array of a cache set.

First, the cache block address is searched. If the access does not hit, it means that the cache block is a unique access. It is added into the array and the counter, which indicates the number of unique cache blocks accessed in that set, is incremented. On a hit, only the recency bits are set to 0. Any policy can be used to manage replacements. We use SRRIP policy. All these operations lie outside the critical path and are independent of the hit/miss activities on the main cache. Finally, it does not require any change to the cache tag array except changing the insertion priority.

Example: Figure 3.3b shows an example of computing Footprint-number. For simplicity, let us assume we sample 4 cache sets and a single application. In the diagram, each array belongs to a separate monitored set. An entry in the array corresponds to the

[2] However, we just use the term Footprint-number throughout.

[3] Only demand accesses update the recency state

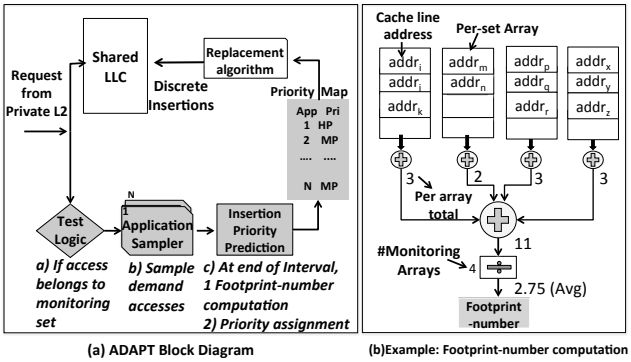


Figure 3.3: (a) ADAPT Block Diagram and b) example for Footprint-number computation

block address that accessed the set. We approximate Footprint-number by computing the average from all the sampled sets. In this example, the sum of all the entries from all the four arrays is 11. And, the average is 2.75. This is the Footprint-number for the application. In a multi-core system, there are as many instances of this component as the number of applications in the system.

3.3.2 Footprint-number based Priority assignment

Like prior studies [JTSE10, WJH+11, SMKM12, Int16], we use 2 bits per cache line to represent re-reference prediction value (RRPV). RRPV '3' indicates the line will be reused in the distant future and hence, a cache line with RRPV of 3 is a candidate for eviction. On hits, only the cache line that hits is promoted to RRPV 0, indicating that it will be reused immediately. On insertions, unlike prior studies, we explore the option of assigning different priorities (up to 4) for applications leveraging the Footprint-number metric.

We propose an insertion-priority-prediction algorithm that statically assigns priorities based on the Footprint-number values. The algorithm assumes that the LLC associativity is 16. However, it still works for larger associative caches as we show later. Table 3.1 summarizes the insertion priorities for each classification. Experiments are performed by varying the high-priority range between [0,3] and [0,8] (6 different ranges), keeping the low-priority range unaffected. Similarly, by keeping the high-priority range [0,3] constant, we change the low-priority range between (7,16) to (12,16) (6 different ranges). In total, from 36 different experiments we fix the priority-ranges. A dynamic approach that uses run-time information to assign priorities is more desirable. We defer this approach to future work. Priority assignments are as follows:

High Priority: All applications in the Footprint-number range [0,3] (both included) are assigned high-priority. When the cache lines of these applications miss, they are inserted with RRPV 0.

Intuition:Applications in this category have working sets that fits perfectly within the cache. Typically, the cache lines of these applications have high number of reuses. Also,

Table 3.1: Insertion Priority Summary

Priority Level	**Insertion Value**
High (HP)	0
Medium (MP)	1 but 1/16th insertion at LP
Low (LP)	2 but 1/16th at MP
Least (LstP)	Bypass but insert 1/32nd at LP

when they share the cache, they do not pose problems to the co-running applications. Hence, they are given high-priority. Inserting with priority 0 allows the cache lines of these applications to stay in the cache for longer periods of time before being evicted.

Medium Priority: All applications in the Footprint-number range (3,12] (3 excluded and 12 included) are assigned medium priority. Cache lines of the applications in this category are inserted with value 1 and rarely inserted with value 2.
Intuition: Applications under this range of Footprint-number have working set larger than the high-priority category however, fit within the cache. From analysis, we observe that the cache lines of these applications generally have moderate reuse except few applications. To balance mixed reuse behavior, one out of the sixteenth insertion goes to low priority 2, while inserted with medium priority 1, otherwise.

Low Priority: Applications in the Footprint-number range (12,16) are assigned low priority. Cache lines of these applications are generally inserted with RRPV 2 and rarely with medium priority 1 (1 out of 16 cache lines).
Intuition: Applications in this category typically have mixed access patterns : $(\{a1, a2\}^k$ $\{s1, s2, s3..sn\}^d)$ with k and d sufficiently small and k slightly greater than d, as observed by TA-DRRIP[JTSE10]. Inserting the cache lines of these applications with low priority 2 ensures (i) cache lines exhibiting low or no reuse at all get filtered out quickly and (ii) cache lines of these applications have higher probability of getting evicted than high and medium priority applications[4].

Least Priority: Applications with Footprint-number range (>= 16) are assigned least priority. Only one out of thirty-two accesses are installed at the last level cache with least priority 3. Otherwise, they are bypassed to the private Level 2. Intuition: Essentially, these are applications that either exactly fit in the cache (occupying all sixteen ways) or with working sets larger than the cache. These applications are typically memory-intensive and when run along with others cause thrashing in the cache. Hence, both these type of applications are candidates for least priority assignment. The intuition behind bypassing is that when the cache lines inserted with least priority are intended to be evicted very soon (potentially without reuse), bypassing these cache lines will allow the incumbent cache lines to better utilize the cache. Our experiments confirm this assumption. In fact, bypassing is not just beneficial to ADAPT. It can be used as a performance booster for other algorithms, as we show in the evaluation section.

[4]It means that transition from 2 to 3 happens quicker than 0 to 3 or 1 to 3 thereby allowing HP and MP applications to stay longer in the cache than LP applications.

Table 3.2: Cost on 16MB,16-way LLC

Policy	**Storage cost**	**N=24 cores**
TA-DRRIP	16-bit/app	48 Bytes
EAF-RRIP	8-bit/address	256KB
SHiP	SHCT table&PC	65.875KB
ADAPT	865 Bytes/app	24KB appx

3.3.3 Hardware Overhead

The additional hardware required by our algorithm is the application sampler and insertion priority prediction logic. The application sampler consists of an array and a counter. The size of the array is same as the associativity. From Section 3.3.2, recall that we assign the same priority(least) to applications that exactly fit in the cache as well as the thrashing applications because, on a 16-way associative cache, both classes of applications will occupy a minimum of 16 ways. Hence, tracking 16 (tag) addresses per set is sufficient. The search and insertion operations on the array are very similar to that of a cache set. The difference is that we store only the most significant 10 bits per cache block. Explanation: the probability of two different cache lines having all the 10 bits same is very low: $(1/2^x)/(2^{10}/2^x)$, where x is the number of tag bits. That is, $1/2^{10}$. Even so, there are separate arrays for each monitoring set. Plus, applications do not share the arrays. Hence, 10 bits are sufficient to store the tag address. 2 bits per entry are used for bookkeeping. Additionally, 8 bits are required for head and tail pointers (4 bits each) to manage search and insertions. Finally, a 4-bit counter is used to represent Footprint-number.

Storage overhead per set is 204 bits and we sample 40 sets. Totally, 204 bits $\times 40 = 8160 bits$. To represent an application's Footprint-number and priority, two more bytes (1 byte each) are needed. To support probabilistic insertions, three more counters each of size one byte are required. Therefore, storage requirement per application sampler is $[8160 \text{ bits} + 40 \text{ bits}] = 8200 bits/application$. In other words, 1KB (approximately) per application.

Table 3.2 compares the hardware cost of ADAPT with others. Though ADAPT requires more storage compared to TA-DRRIP [JTSE10], it provides higher performance improvement and is better compared to EAF [SMKM12] and SHiP [WJH$^+$11] in both storage and performance.

3.3.4 Monitoring in a realistic system:

In our study, we assume that one thread per core. Therefore, we can use the core ID for the thread. On an SMT machine, the thread number/ID would have to be transmitted with the request from the core for our scheme to work properly.

If an application migrates (on a context-switch) to another core, the replacement policy applied for that application during the next interval will be incorrect. However, the interval is not long (1Million LLC misses). The correct Footprint-number and

insertion policy will be re-established in the following monitoring interval onward. In data-centers or server systems, tasks or applications are not expected to migrate often. A task migrates only in exceptional cases like shutdown or, any power/performance related optimization. In other words, applications execute(spend) sufficient time on a core for the heuristics to be implemented. Finally, like prior works[QJP+07, JHQ+08, JTSE10, SMKM12, WJH+11, QP06], we target systems in which LLC is organized as multiple banks with uniform access latency.

3.4 Experimental Study

3.4.1 Methodology

For our study, we use BADCO [VMS12] cycle-accurate x86 CMP simulator. Table 3.3 shows our baseline system configuration. We do not enforce inclusion in our cache-hierarchy and all our caches are write-back caches. LLC is 16MB and organized into 4 banks. We model bank-conflicts, but with fixed latency for all banks like prior studies [JTSE10, WJH+11, SMKM12].A VPC[NLS07] based arbiter schedules requests from L2 to LLC. We use DRAM model similar to [SMKM12].

Table 3.3: Baseline System Configuration

Processor Model	4-way OoO, 128 entry ROB, 36 RS, 36-24 entries LD-ST queue
Branch pred.	TAGE, 16-entry RAS
IL1 & DL1	32KB; LRU; next-line prefetch;I$:2-way;D$:8-way; 64 bytes line
L2 (unified)	256KB,16-way, 64 bytes line, DRRIP, 14-cycles, 32-entry MSHR and 32-entry retire-at-24 WB buffer
LLC (unified)	16MB, 16-way, 64 bytes line, TA-DRRIP, 24 cycles, 256-entry MSHR and 128-entry retire-at-96 WB buffer
Main-Memory (DDR2)	Row-Hit:180 cycles, Row-Conflict:340 cycles, 8 banks,4KB row, XOR-mapped[ZZZ00]

Table 3.4: Empirical Classification of Applications

FP-num	L2 MPKI	Memory Intensity
< 16	< 1	VeryLow (VL)
	$[1, 5)$	Low (L)
	> 5	Medium (M)
$>= 16$	< 5	Medium (M)
	$[5, 25)$	High (H)
	> 25	VeryHigh (VH)

Table 3.5: Benchmark classification based on Footprint-number and L2-MPKI.

Name	Fpn(A)	Fpn(S)	L2-MPKI	Type
black	7	6.9	0.67	VL
calc	1.33	1.44	0.05	VL
craf	2.2	2.4	0.61	VL
deal	2.48	2.93	0.5	VL
eon	1.2	1.2	0.02	VL
fmine	6.18	6.12	0.34	VL
h26	2.35	2.53	0.13	VL
nam	2.02	2.11	0.09	VL
sphnx	5.2	5.4	0.35	VL
tont	1.6	1.5	0.75	VL
swapt	1	1	0.06	VL
gcc	3.4	3.2	1.34	L
mesa	8.61	8.41	1.2	L
pben	11.2	10.8	2.34	L
vort	8.4	8.6	1.45	L
vpr	13.7	14.7	1.53	L
fsim	10.2	9.6	1.5	L
sclust	8.7	8.4	1.75	L
art	3.39	2.31	26.67	M
Name	**Fpn(A)**	**Fpn(S)**	**L2-MPKI**	**Type**
bzip	4.15	4.03	25.25	M
gap	23.12	23.35	1.28	M
gob	16.8	16.2	1.28	M
hmm	7.15	6.82	2.75	M
lesl	6.7	6.3	20.92	M
mcf	11.9	12.4	24.9	M
omn	4.8	4	6.46	M
sopl	10.6	11	6.17	M
twolf	1.7	1.6	16.5	M
wup	24.2	24.5	1.34	M
apsi	32	32	10.58	H
astar	32	32	4.44	H
gzip	32	32	8.18	H
libq	29.7	29.6	15.11	H
milc	31.42	30.98	22.31	H
wrf	32	32	6.6	H
cact	32	32	42.11	VH
lbm	32	32	48.46	VH
STRM	32	32	26.18	VH

3.4.2 Benchmarks

We use benchmarks from SPEC 2000 and 2006 and PARSEC benchmark suites, totaling 36 benchmarks (31 from SPEC and 4 from PARSEC and 1 Stream benchmark). Ta-

Table 3.6: Workload Design

Study	#Workloads	Composition	#Instructions
4-core	120	Min 1 thrashing	1.2B
8-core	80	Min 1 from each class	2.4B
16-core	60	Min 2 from each class	4.8B
20-core	40	Min 3 from each class	6B
24-core	40	Min 3 from each class	7.2B

ble 5.2 shows the classification of all the benchmarks and Table 3.4 shows the empirical method used to classify memory intensity of a benchmark based on its Footprint-number and L2-MPKI when run alone on a 16MB, 16-way set-associative cache. In Table 5.2, the column Fpn(A) represents Footprint-number value obtained by using all sets while the column Fpn(S) denotes Footprint-number computed by sampling. Only vpr shows > 1 difference in Footprint-number values. Only to report the upper-bound on the Footprint-numbers, we use 32-entry storage. In our study, we use only 16-entry array. We use a selective portion of 500M instructions from each benchmark. We warm-up all hardware structures during the first 200M instructions and simulate the next 300M instructions. If an application finishes execution, it is re-executed until all applications finish.

3.4.3 Workload Design

Table 3.6 summarizes our workloads. For 4 and 8-core workloads, we study with 4MB and 8MB shared caches while 16, 20 and 24-core workloads are studied with a 16MB cache since we target caches where $\#applications \geq \#llcassociativity$.

3.5 Results and Analysis

3.5.1 Performance on 16-core workloads

Figure 3.4 shows performance on the weighted-speedup metric over the baseline TA-DRRIP and three other state-of-the-art cache replacement algorithms. We evaluate two versions of ADAPT: one which inserts all cache lines of least priority applications (referred as ADAPT_ins) and the version which mostly bypasses the cache lines of least priority applications (referred as ADAPT_bp32). Our best performing version is the one that bypasses the cache lines of thrashing applications. Throughout our discussion, we refer to ADAPT as the policy that implements bypassing. From Figure 3.4, we observe that ADAPT consistently outperforms other cache replacement policies. It achieves up to 7% improvement with 4.7% on average. As mentioned in Section 3.2, with set-dueling, applications with working-set larger than the cache, implement SRRIP policy, which causes higher contention and thrashing in the cache. Similarly, SHiP learns the reuse behavior of region of cache lines (grouped by their PCs) depending on the hit/miss behavior. A counter records the hits (indicating intermediate) and misses

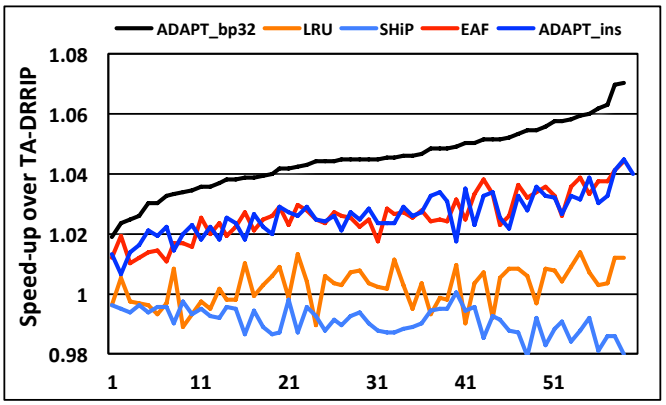


Figure 3.4: Performance of 16-core workloads

(indicating distant) reuse behavior for the region of cache lines. Since SHiP implements SRRIP, it observes similar hit/miss pattern as TA-DRRIP for thrashing applications. Consequently, like TA-DRRIP, it implements SRRIP for all applications. Only 3% of the misses are predicted to have distant reuse behavior. The marginal drop in performance (1.1%appx) is due to inaccurate distant predictions on certain cache-friendly applications. Overall, ADAPT uses Footprint-number metric to efficiently distinguish across applications.

LRU inserts the cache lines of all applications at MRU position. However, cache-friendly applications only partially exploit such longer most-to-least transition time because the MRU insertions of thrashing applications pollute the cache. On the other hand, ADAPT efficiently distinguishes applications. It assigns least priority to thrashing applications and effectively filter out their cache lines, while inserting recency-friendly applications with higher priorities, thus achieving higher performance.

The EAF algorithm filters recently evicted cache addresses. On a cache miss, if the missing cache line is present in the filter, the cache line is inserted with near-immediate reuse (RRPV 2). Otherwise, it is inserted with distant reuse (RRPV 3). In EAF, the size of the filter is such that it is able to track as many misses as the number of blocks in the cache (that is, working-set twice the cache). Hence, any cache line that is inadvertently evicted from the cache falls in this filter and gets intermediate reuse prediction. Thus, EAF achieves higher performance compared to TA-DRRIP, LRU and SHiP. Interestingly, EAF achieves performance comparable to ADAPT_ins. On certain workloads, it achieves higher performance while on certain workloads it achieves lesser performance. This is because, with ADAPT (in general), applications with smaller Footprint-number are inserted with RRPV 0 or 1. But, when such applications have poor reuse, EAF (which inserts with RRPV 2 for such applications) filters out those cache lines. On the contrary, applications with smaller Footprint-number but moderate or more number of reuses, gain from ADAPT's discrete insertions. Nevertheless, ADAPT (with bypassing) consistently outperforms EAF algorithm. We observe that the presence of thrashing applications causes the filter to get full frequently. As a result EAF is only able to

partially track the application's (cache lines). On the one hand, some cache lines of non thrashing (recency-friendly) that spill out of the filter get assigned a distant (RRPV 3). On the other hand, cache lines of the thrashing applications that occupy filter positions get intermediate (RRPV 2) assignment.

3.5.2 Impact on Individual Application Performance

We discuss the impact of ADAPT on individual application's performance. The results are averaged from all the sixty 16-core workloads. Only applications with change (>= 3%) in MPKI or IPC are reported. From Figures 3.5 & 3.6, we observe that bypassing does not cause slow-down (except cactusADM) on least priority applications and provides substantial improvement on high and medium priority applications. Therefore, our assumption that bypassing most of the cache lines of the least-priority applications to be beneficial to the overall performance is confirmed. As we bypass the cache lines (31/32 times) of the least-priority applications (instead of inserting), the cache state is not disturbed most of the times: cache lines which could benefit from staying in the cache remain longer in the cache without being removed by cache lines of the thrashing applications. For most of the applications bypassing provides substantial improvement in MPKI and IPC, as shown in Figure 3.6. Bypassing affects only cactusADM. This is because some of their cache lines are reused immediately after insertion. For gzip and lbm, though MPKI increases, they do not suffer slow-down in IPC. Because, an already memory-intensive application with high memory-related stall time, which when further delayed, does not experience much slow-down[MM08].

3.5.3 Impact of Bypassing on cache replacement policies

In this section, we show the impact of bypassing distant priority cache lines instead of inserting them on all replacement policies. Since LRU policy inserts all cache lines with MRU (high) priority, there is no opportunity to implement bypassing. From Figure 3.7, we observe that bypassing achieves higher performance for replacement policies except SHiP. As mentioned earlier, SHiP predicts distant reuse only for 3% of the cache lines. Of them, 69% (on average) are miss-predictions. Hence, there is minor drop in performance.

On the contrary, TA-DRRIP, which implements bi-modal(BRRIP) on certain cache sets, bypasses the distant priority insertions directly to the private L2 cache, which is beneficial. Consequently, it learns BRRIP for the thrashing applications. Similarly, EAF with bypassing achieves higher performance. EAF, on average, inserts 93% of its cache lines with distant reuse prediction providing more opportunities to bypass. However, we observe that 33% (appx) of distant reuse predictions are incorrect[5]. Overall, from Figure 3.7, we can make two conclusions: first, our intuition of bypassing distant reuse

[5]Miss-predictions are accounted by tracking distant priority (RRPV 3) insertions which are not reused while staying in the cache, but referenced (within a window of 256 misses per set) after eviction. Here, we do not account distant priority insertions that are reused while staying in the cache because such miss-predictions do not cause penalty.

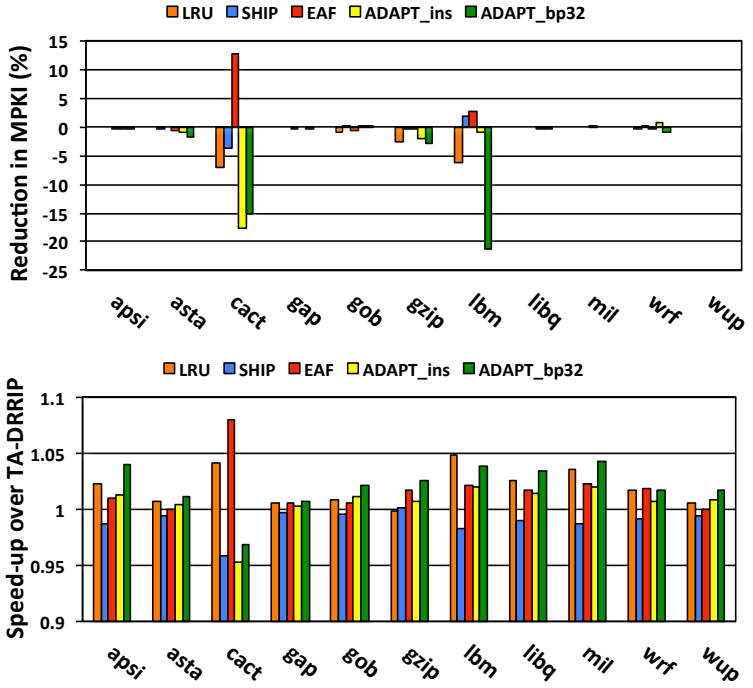


Figure 3.5: MPKI(top) and IPC(below) of thrashing applications

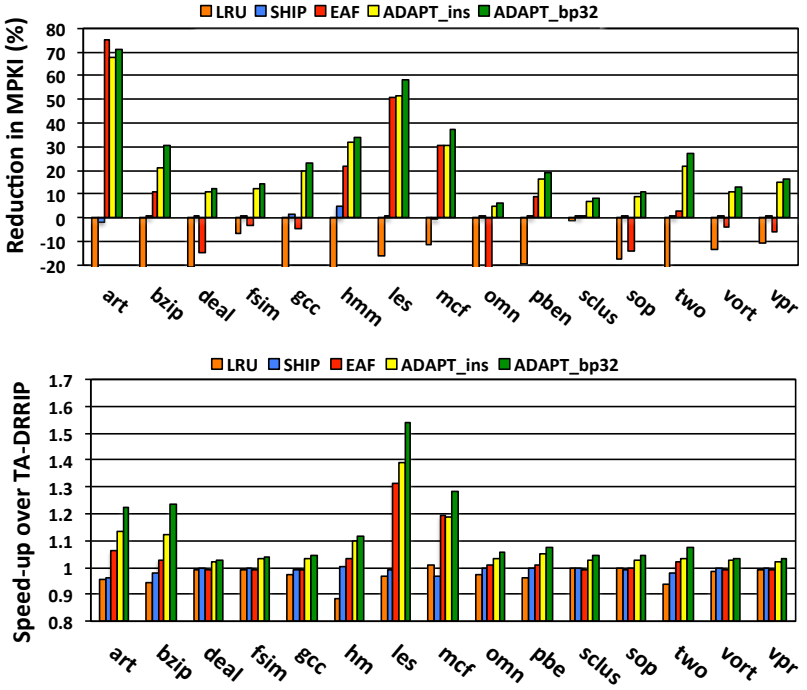


Figure 3.6: MPKI(top) and IPC(below) of non-thrashing applications

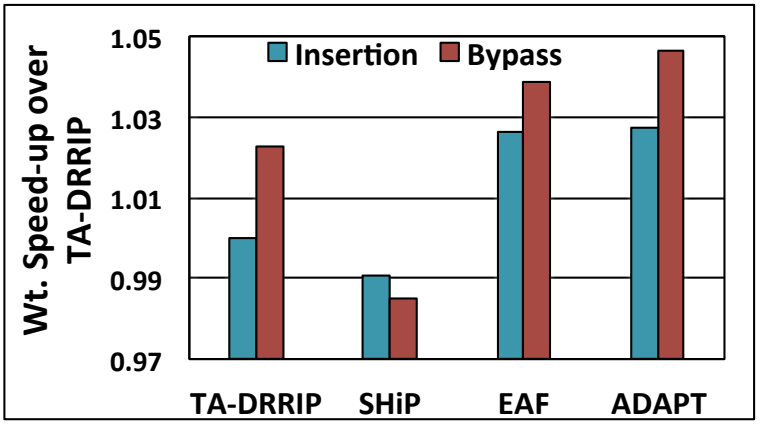
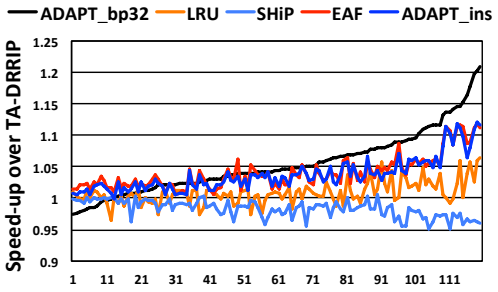


Figure 3.7: Impact of Bypassing on replacement policies

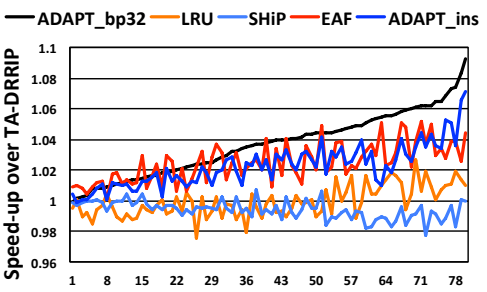
cache lines can be applied to other replacement policies. Second, Footprint-number is a reliable metric to approximate an application's behavior: using Footprint-number, ADAPT distinguishes thrashing applications and bypasses their cache lines.

3.5.4 Scalability with respect to number of applications

In this section, we study how well ADAPT scales with respect to the number of cores sharing the cache. Figures 3.8 and 3.9 show the s-curves of weighted speed-up for 4,8,20 and 24-core workloads. ADAPT outperforms prior cache replacement policies. For 4-core workloads, ADAPT yields average performance improvement of 4.8%, and 3.5% for 8-core workloads. 20 and 24-core workloads achieve 5.8% and 5.9% improvement, on average, respectively. Here, 20 and 24-core workloads are studied on 16MB,16-way associative cache. Recall our proposition : $(\#cores \geq associativity)$.

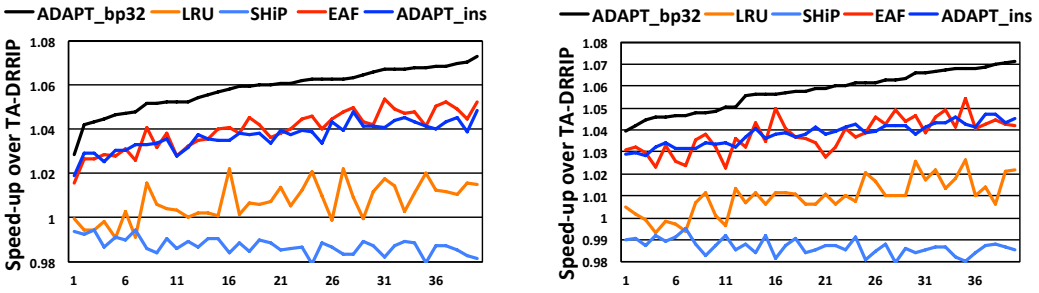


(a) 4-core (120 workloads)



(b) 8-core (80 workloads)

Figure 3.8: Performance of ADAPT with respect to number of applications for 4 and 8-cores



(a) 20-core (40 workloads) (b) 24-core (40 workloads)

Figure 3.9: Performance of ADAPT with respect to number of applications for 20 and 24-cores

3.5.5 Sensitivity to Cache Configurations

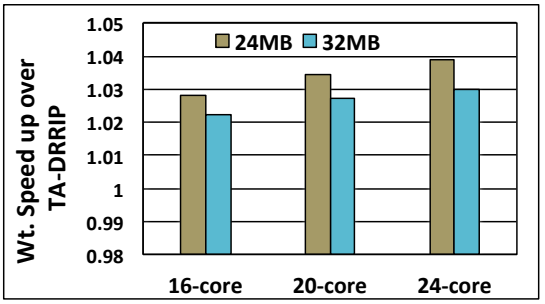


Figure 3.10: Performance on Larger Caches

In this section, we study the impact of ADAPT replacement policy on systems with larger last level caches. In particular, the goal is to study if Footprint-number based priority assignment designed for 16-way associative caches applies to larger associative (> 16) caches as well. For 24MB and 32MB caches, we increase only the associativity of the cache set from 16 to 24 and 16 to 32, respectively. Certain applications still exhibit thrashing behaviors even with larger cache sizes which ADAPT is able to manage and achieve higher performance on the weighted Speed-up metric(Figure 3.10).

3.6 Conclusion

Future multi-core processors will continue to employ shared last level caches. However, their associativity is expected to remain in the order of sixteen consequently posing two new challenges: (i) the ability to manage more cores (applications) than associativity and (ii) the replacement policy must be application aware and allow to discretely (> 2) prioritize applications. Towards this end we make the following contributions:

•We identify that existing approach of observing hit/miss pattern to approximate applications' behavior is not efficient.
•We introduce the Footprint-number metric to dynamically capture the working-set size of applications. We propose Adaptive Discrete and de-prioritized Application Prioritization (ADAPT), a new cache replacement algorithm, which consists of a monitoring mechanism and an insertion-priority-prediction algorithm. The monitoring mechanism dynamically captures the Footprint-number of applications on an interval basis. The prediction algorithm computes insertion priorities for applications from the Footprint-numbers under the assumption that smaller the Footprint-number, better the cache utilization. From experiments we show ADAPT is efficient and scalable ($(\#cores \geq \#associativity)$).

Chapter 4

Band-pass Prefetching : A Prefetch-fraction driven Prefetch Aggressiveness Control Mechanism

In this chapter, we present our second contribution, Band-pass Prefetching. We first motivate our work by discussing the problem associated with the state-of-the-art prefetcher aggressiveness control mechanism namely, Hierarchical Prefetcher Aggressiveness Control (HPAC) [EMLP09] and CAFFEINE [PB15]. Then, we describe our proposed mechanism, followed by evaluation and comparison against state-of-the-art mechanisms.

4.1 Introduction

An aggressive hardware prefetcher may completely hide the latency of off-chip memory accesses. However, it may cause severe interference at the shared resources (last level cache and memory bandwidth) of a multicore system [EMLP09, ELMP11, WJM$^+$11, SYX$^+$15, PB15, JBB$^+$15, Pan16, LMNP08, LS11, BIM08]. To manage prefetching in multicore systems, prior studies [SMKP07, EMLP09, ELMP11, PB15, Pan16] have been proposed to dynamically control (also known as throttling) the prefetcher aggressiveness by adjusting the prefetcher-configuration at runtime. These mechanisms make dynamic throttling decisions by computing several parameters such as prefetch-accuracy, lateness, prefetcher-caused interference at the last level cache and DRAM in the form of pollution, row-buffer, bus and bank conflicts. Carefully tuned threshold values of these parameters drive prefetcher aggressiveness control decisions.

Problem: Prior works such as Hierarchical Prefetcher Aggressiveness Control (HPAC) [EMLP09] and CAFFEINE [PB15] do not completely alleviate the problem of prefetcher-caused interference in multicore systems. With HPAC, we observe that the use of multiple metrics (driven by their threshold values) does not capture the actual interference in the system, and in most cases leads to incorrect throttling decisions. With CAFFEINE, approximate estimation of the average last level cache miss penalty leads to biased throttling decisions overlooking interference caused due to prefetchers. Alto-

gether, prior works still provide scope for performance improvement.

Solution: We propose a solution to manage interference caused by prefetchers in multicore systems. Our solution builds on two observations. First, for a given application, fewer the number of prefetch requests generated, less likely that they are useful. That is, a strong positive correlation exists between the accuracy of a prefetcher and the amount of prefetch requests it generates for an application relative to its total prefetch and demand requests. Second, more the aggregate number of prefetch requests in the system, higher the miss penalty on the demand misses at last level cache. That is, service time of the demand misses at the last level cache increases with the increase in the aggregate number of prefetch requests (misses)[1] that also leave the last level cache. In particular, we observe a strong positive correlation between the ratio of average miss service times of demand to prefetch misses and the ratio of aggregate prefetch to demand misses at shared the LLC-DRAM interface.

Based on the two observations, we use the concept of prefetch-fraction to infer the (i) usefulness (in terms of prefetch-accuracy) of prefetching to an application and (ii) interference caused by a prefetcher at the shared LLC-DRAM interface. We define prefetch-fraction of an application as the fraction of L2 prefetch requests a prefetcher generates for an application with respect to its total requests (demand misses, L1 and L2 prefetch requests). To infer the usefulness of prefetching to an application, we compute prefetch-fraction for each application independently at the private L2 caches (that is, at the interface between private L2s and shared LLC). To infer interference due to a prefetcher, we compute prefetch-fraction for each application at the shared LLC-DRAM interface.

Based on the inference drawn from these parameters, we apply simple prefetcher throttling at two levels. First, at the private L2 (application) level when the inferred prefetch-accuracy is low. Second, at the shared LLC-DRAM interface (globally), when prefetch requests are likely to delay demand misses. The two mechanisms independently control the flow of prefetch requests between a range of prefetch-to-demand ratios. This is analagous to Band-pass filtering in signal processing systems [OWN96]. A band-pass filter consists of high-pass and low-pass components: high-pass allows signal frequencies that are only higher than a threshold value, while low-pass allows only signal frequencies that are lower than a threshold value. Together, the two filters allow only a band of signal frequencies to pass through. Similarly, our two mechanisms allow only prefetch requests that are between a range of prefetch-to-demand ratios to flow through from LLC to DRAM. Hence, we refer to our solution as Band-pass prefetch filtering or simply, Band-pass prefetching.

4.2 Background

This section provides a background on our baseline system and the definitions used throughout the paper. We then briefly describe HPAC [EMLP09] and CAFFEINE

[1]By prefetch misses, we refer to the L2 prefetch requests generated by the prefetcher sitting beside each private L2 cache that miss and leave LLC for DRAM access.

[PB15], two state-of-the-art dynamic prefetcher aggressiveness control mechanisms.

4.2.1 Baseline Assumptions and Definitions

In this paper, our goal is to propose a mechanism that can manage prefetcher-caused interference in multicore system, and not to propose a new prefetching mechanism itself. Throughout this chapter, we consider a system with a cache-hierarchy of three levels with private L1 and L2 caches. The last level cache (LLC) is shared by all the cores. L1 caches feature a next-line prefetcher while L2 features a *stream prefetcher which we intend to control*. Our stream prefetcher model is closer to the implementations of Feedback Directed Prefetching (FDP) [SMKP07] and IBM Power series of processors[SKS+11]. It sits beside L2 and trains on L2 misses and L2 prefetch-hits. Only one stream entry (a unique prefetchable context) is allocated per 4KB page entry. It tracks 32 outstanding streams and issues prefetch requests with prefetch-distance of 8 and prefetch-degree of 4.

Definitions. Throughout this chapter, we use the following terminologies: Prefetch-distance: It is the number of cache lines ahead of X that the prefetcher tries to prefetch, where X is the cache block address of the cache miss that allocated the current stream. Prefetch-degree: It is the number of prefetch requests issued when there is an opportunity to prefetch. Throttle-up/down: A prefetcher's aggressiveness is defined in terms of its configuration : prefetch-distance and degree. Throttling-up/down refers to increasing/decreasing the values of prefetch-distance and degree.

4.2.2 Problem with the state-of-the-art Mechanisms

HPAC: HPAC consists of a per-core local and a shared global feedback components. While HPAC's local component (FDP[SMKP07]) attempts to maximize the benefit of prefetching to an application, the global component attempts to minimize the interference caused by a prefetcher. The local component computes prefetch-accuracy, lateness and pollution metrics local to an application. The global component computes interference related parameters such as bandwidth consumed (in cycles) by prefetch requests, amount of time (bandwidth needed in cycles) demands of an application wait because of prefetch requests for a memory resource (BWN) and prefetcher-caused cache pollution (POL). For each application, HPAC also computes BWNO metric, which is the bandwidth requirement of other cores. HPAC assumes that BWNO of a prefetcher becomes high when its prefetch requests consume high bandwidth (BWC), and forces memory requests of other applications to wait. Based on the threshold values of all these metrics, HPAC's global component infers an application to be interfering-with-others or not. If a prefetcher is found to be interfering, HPAC's global control throttles it down. Otherwise, it allows the decision of its local component (FDP).

Problem with HPAC: The issue with HPAC is the use of multiple metrics and inference drawn from them. A given value of a metric does not reflect the run-time behavior of an application due to interference caused when large number of applications run on the system. For example, a prefetcher's accuracy drops down when its prefetch requests

are delayed at the shared resources by the co-running applications. Similarly, HPAC uses bandwidth needed by others (BWNO) parameter to account the bandwidth requirement of all other applications in the system, except the one under consideration. When large applications run on the system, BWNO of an application tends to be higher while its prefetcher may not consume much bandwidth (BWC). Under the scenario in which an application's prefetch-accuracy is low and BWNO parameter is high, HPAC infers the application to be interfering-with-others and decides to throttle-down its prefetcher. In contrast, an application with high-accuracy is throttled-up although its prefetch requests consume high bandwidth. We observe several instances of such scenarios where HPAC does not capture interference and makes incorrect throttling decisions.
CAFFEINE: CAFFEINE takes a fine-grained account of interference caused by prefetch requests at each of the shared resources, such as DRAM bus, banks, row-buffers and shared last level cache. It accounts benefit of prefetching to an application by estimating the amount of cycles saved in terms of its off-chip memory accesses. It normalizes both interference and prefetch usefulness to a common scale of processor cycles, which it refers to as net-utility of a prefetcher. It uses both system-wide and per-core net-utilities to make throttling decisions. In particular, CAFFEINE throttles-up prefetcher when the system-wide net-utility is positive and throttles them down, otherwise.
Problem with CAFFEINE: CAFFEINE estimates the average last level cache miss-penalty by accumulating the latency of individual memory requests (difference in arrival and start times) and then, computes the arithmetic-mean on this accumulated sum of latencies over all requests. The resulting mean value is approximated as the average miss-penalty. In doing so, CAFFEINE treats each memory request as an isolated event and does not take into account overlapping memory accesses inherent in applications. Therefore, miss-penalty is overestimated, which when used in its utility model, overestimates the off-chip memory access cycles saved due to prefetching. Hence, CAFFEINE's throttling decisions favor aggressive prefetching.

4.3 Motivational Observations

In the following paragraphs, we discuss how Prefetch-fraction statistically captures both the usefulness(prefetch-accuracy) of prefetching and prefetch-caused interference (delay induced on demands by prefetch requests) at the shared memory bandwidth.

4.3.1 Correlation between Prefetch-accuracy and Prefetch-fraction

The amount of prefetch requests generated by a stream prefetcher and hence, its usefulness depends on an application's memory access behavior. In particular, usefulness (in terms of prefetch-accuracy) of prefetching depends on the fraction of L2 prefetch requests generated with respect to an application's total requests. Figures 4.1 and 4.2 illustrate the correlation between L2 prefetch-fraction and L2 prefetch-accuracy for the baseline aggressive stream prefetcher and for Feedback Directed Prefetching (FDP), respectively. FDP is a state-of-the-art single-core prefetcher aggressiveness control engine.

From Figure 4.1, it can be observed that for the baseline aggressive stream

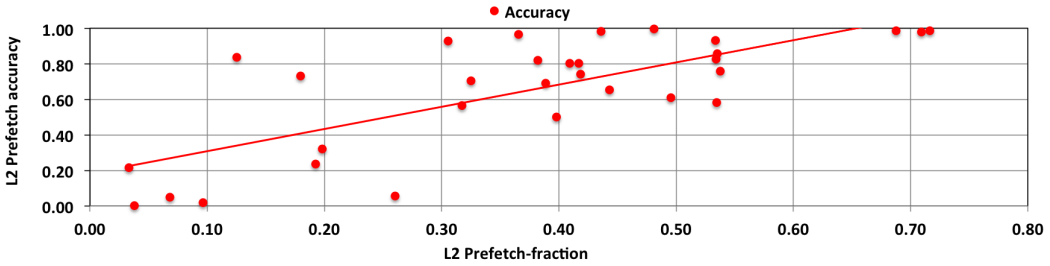


Figure 4.1: Scatter plot showing positive correlation between L2 Prefetch-fraction versus L2 Prefetch-accuracy for the baseline aggressive prefetching: Pearson correlation coefficient: 0.76 and Spearman rank correlation: 0.68.

prefetcher, L2 prefetch-fraction varies across applications. For applications like astar, bzip, milc and omnet, the prefetcher generates fewer prefetch requests than for applications with streaming behavior such as apsi, libq, leslie, lbm, wupwise and stream benchmark. For astar, bzip, milc and omnet, L2 prefetch-fraction is less than 10% and their L2 prefetch-accuracy is also low (around 5%). However, with increase in L2 prefetch-fraction values (along x-axis), L2 prefetch-accuracy also increases. A linear plot across all the data points in the figure shows a positive correlation. In particular, the plot shows 0.76 on the Pearson correlation coefficient [Sha05] and 0.68 on the Spearman rank-correlation coefficient metric [Sha05]. A similar observation can be made from Figure 4.2 for FDP. We observe similar correlation relationship between L2 prefetch-fraction and L2 prefetch-accuracy for Access Map Pattern Matching, AMPM prefetcher [IIH09]. AMPM creates an access map of all the cache lines of the pages it tracks and issues prefetch requests by storing state information. The state information decides to issue a prefetch or not. Essentially, AMPM prefetcher is orthogonal to stream prefetching. Our observation holds good for AMPM prefetcher as well. Therefore, we conclude that there is a strong positive correlation between L2 prefetch-fraction and L2 prefetch-accuracy: lower the fraction of prefetch requests generated, less-likely that

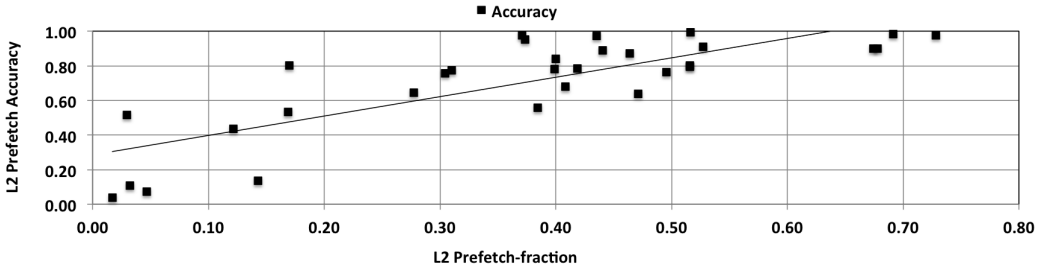


Figure 4.2: Scatter plot showing positive correlation between L2 Prefetch-fraction versus L2 Prefetch-accuracy for Feedback directed prefetching: Pearson correlation coefficient: 0.80 and Spearman rank correlation: 0.75.

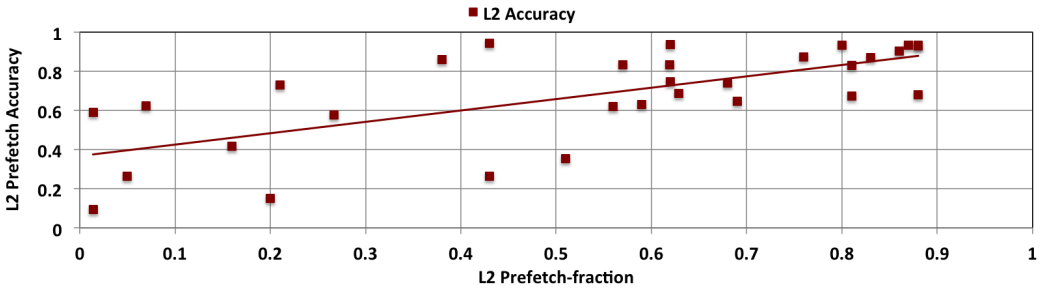


Figure 4.3: Scatter plot showing positive correlation between L2 Prefetch-fraction versus L2 Prefetch-accuracy for Access Map Pattern Matching prefetching: Pearson correlation coefficient: 0.68 and Spearman rank correlation: 0.65.

they are useful. Therefore, we approximate usefulness of prefetching (prefetch-accuracy) using L2 prefetch-fraction metric.

4.3.2 Correlation between Prefetcher-caused delay and Prefetch-fraction

High performance memory controllers like First Ready-First Come First Serve (FR-FCFS) [RDK+00] and Prefetch-Aware DRAM Controller (PADC) [LMNP08] re-order requests to exploit row-buffer locality, and maximize throughput. When a memory controller prioritizes row-hits over row-conflicts, prefetch requests tend to get prioritized over demands. Because, an earlier request opens a row and the subsequent sequence of prefetch requests to the same row exploit row-buffer locality. Therefore, the average service time (LLC miss-penalty or roundtrip latency between LLC and DRAM) of prefetch requests is shorter than that of demands. This disparity in the service times between prefetch and demand requests grows linearly with increase in the ratio of total prefetches to that of total demand requests at the LLC-DRAM interface.

Figure 4.4 illustrates this observation for a 16-core workload that consists of applications such as vpr, stream-cluster, wup, mcf, blackscholes, hmmer, stream, lbm, apsi, sphinx, leslie, mesa, vort, pben, astar and wrf which have mixed prefetch-friendliness characteristics (Refer Table 5.2). The x-axis represents execution time in intervals of 1 Million LLC misses, and the y-axis represents (i) the ratio of total prefetches to that demands as well as (ii) the ratio of average miss service times of demands to prefetch requests. Under aggressive stream-prefetching that uses no prefetcher throttling, the total prefetches at the LLC-DRAM interface is always higher than that of total demands.

From Figure 4.4, we observe that the ratio of average miss service times of demands to prefetch requests increases and decreases with the increase and decrease in the ratio of total prefetch requests to total demands. In other words, as the ratio of total prefetches to total demands increases, the degree of interference induced on demands (observed in terms of average LLC miss service times of demands) by prefetch requests also increases. Statistically, we observe a very strong positive correlation (0.97

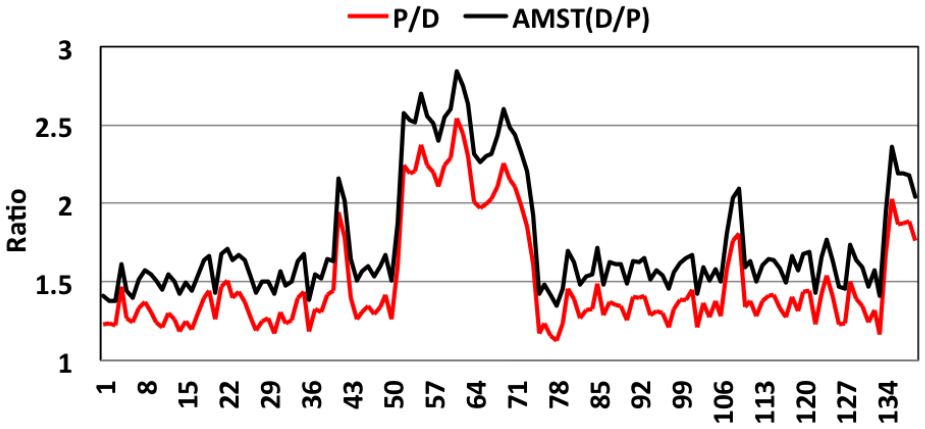


Figure 4.4: Ratio of LLC miss service times of demand to prefetch requests increases with increase in the ratio of total prefetch requests to that of demands in the system. AMST : Average Miss Service Time.

on Pearson's coefficient[2]) between ratios of the two quantities. We also observe (i) a strong positive correlation (0.96 on Pearson's coefficient) between the ratio of aggregate prefetch to demand requests and the ratio of bandwidth consumed by prefetch to demands and (ii) a strong correlation (0.95 on Pearson's coefficient) between the ratio of bandwidth consumed by prefetch to demands and the ratio of average service times of demand to prefetch requests. However, estimation of latency gives a direct indication on prefetcher-caused interference, we use it in our study. From these two observations, we therefore conclude that the interference caused by prefetch requests on demands can be approximated using the ratio of aggregate prefetch to demand requests at the LLC-DRAM interface.

Altogether, prefetch-fraction, as a metric, captures both the usefulness of prefetching (in terms of prefetch-accuracy) to an application when measured at the L2-LLC interface as well as the prefetcher-caused interference (in terms of prefetcher-induced delay) when measured at LLC-DRAM interface.

4.4 Band-pass prefetching

In this section, we present Band-pass prefetching, a dynamic prefetcher aggressiveness control mechanism to manage prefetcher-caused interference in multicore systems, which exploits the two correlations we discussed in Section 4.3.

4.4.1 High-pass Prefetch Filtering

In Section 4.3.1, we showed that L2 prefetch-accuracy strongly correlates with L2 prefetch-fraction. To leverage this correlation, we compute prefetch-fraction for an

[2]We obtain correlation from all the 16-core workloads.

application at run-time. If the measured value of prefetch-fraction is less than certain threshold, the component probabilistically issues/allows prefetch requests to go to next level. That is, once in every sixteenth prefetch request is issued to the next level; the rest of the prefetch requests are dropped[3]. Since this filter component issues all the generated prefetch requests to the next level only when prefetch-fraction is higher than the threshold, we call this component High-pass prefetch filter (analogous to high-pass filter in signal processing).

4.4.1.1 Measuring Prefetch-fraction

For measuring prefetch-fraction, we use two counters: L2PrefCounter and TotalCounter. L2PrefCounter records the L2 prefetch requests while TotalCounter holds the total requests (demands and prefetches from L1 and L2 caches) at the L2-LLC interface. At the end of an interval, the ratio of the two counters gives the L2 prefetch-fraction value, which is stored in another register called Prefetch-fraction register. Only the counters are reset; Prefetch-fraction register's value is used to make prefetch issue decisions for the next interval.

It is to be noted that L2PrefCounter is incremented whenever a prefetch request is generated[4] by the prefetcher ; not when a prefetch request is issued to the next level. Doing so has two implications: First we are interested in the number of prefetch requests the prefetcher is able to generate for the application and hence, infer the usefulness of prefetching. Secondly, incrementing the L2PrefCounter only when a prefetch request is issued, causes positive-feedback on prefetch-filtering: when an application undergoes a phase change in which the prefetcher generates only fewer number of prefetch requests, L2PrefCounter records a smaller value of L2 prefetch requests. Consequently, the measured value of prefetch-fraction becomes low and the filter decides to partially(1 out of 16) issue prefetch requests in the subsequent interval. When there is another phase change in that application where prefetcher is able to generate more prefetch requests than the earlier (ramped-down phase), the counter still keeps recording only few number of prefetch requests being issued. Consequently, the High-pass filter issues only fewer number of prefetch requests though the prefetcher generates a large fraction of them. Either it takes multiple intervals to recover back to the steady state or the filter never adapts to the phase changes of applications.

4.4.2 Low-pass Prefetch Filtering

In a multi-core system, memory requests of one application interfere with the others at the shared last level cache and memory access. We have observed that the LLC miss

[3]By dropping a prefetch request, we refer to not issuing it to the next level (from L2 to LLC). We drop prefetch requests instead of adjusting the prefetcher-configuration in distance and degree. We observe dropping prefetch requests performs better than the latter because, dropping reduces prefetch issue-rate quicker and also issues fewer prefetch requests.

[4]We use the term *generate* to refer to the output of the prefetcher while we use *issue* to refer prefetch requests that are actually sent for fetching data from the memory.

service time of demand requests (and therefore the likely stall-time of the missing processor) increases with the number of prefetch requests. Ideally, we expect the average service time of demand requests to be less than that of prefetch requests because demands are likely to stall the processor as compared to prefetch requests. Therefore, we propose a filter at the shared LLC-DRAM interface that controls prefetcher aggressiveness when the average miss service time of demands exceeds that of prefetch requests. However, testing this condition alone is not sufficient because the ratio of prefetch to demand requests and their relative bandwidth consumption are also strongly correlated (recall from Section 4.3 a correlation of 0.96 on Pearson's metric). Therefore, controlling the prefetcher aggressiveness only by comparing the ratio of average miss service times of demands and prefetches alone can lead to conservatively throttling prefetchers while the prefetch requests do not consume much bandwidth (and not cause interference). Therefore, our mechanism also checks if the total prefetches exceed the demands when the average service time of demand requests exceed that of prefetch requests. Altogether, the condition to apply prefetcher aggressiveness control is given by Equation 4.1, where AMST(D or P) refers to Average Miss Service Time of demand or prefetch requests. TP and TD refers to the total prefetches and demands at the LLC-DRAM interface, respectively.

$$\left(\left(\frac{\text{AMST}(D)}{\text{AMST}(P)} > 1\right) \,\&\, \left(\frac{TP}{TD} > 1\right)\right) \tag{4.1}$$

Here, in Equation 4.1, we make an approximation on TP/TD. Since TP/TD strongly correlates with BWCP/BWCD, the relationship between them is

$$\text{TP/TD} = F(BWCP/BWCD)$$

We find this relationship as ($TP/TD = \alpha \times$ (BWCP/BWCD)), where (α) is around 1. While we explore across different values, it is hard to fix the exact value. Hence, we approximate it to 1 and check only the ratio ($TP/TD > 1$) alone, which is very simple to implement in HW: a 16-bit comparator.

In the following subsection, we explain our mechanism that estimates the average miss service times of demands and prefetches followed by how we collect prefetch-fraction metric for applications at the shared LLC-DRAM interface.

4.4.2.1 Estimation of Average Miss Service Time

We propose a mechanism that uses a set of counters and a comparator logic to estimate the average service times of demand and prefetch requests. Table 4.1 lists the set of counters and their purpose. Algorithm 1 describes our mechanism. Its explanation is as follows:
Explanation: The algorithm is triggered either on a miss[5] at the LLC or when a miss is

[5]In this subsection, by miss we either refer prefetch or demand miss alone. Since we are only interested in their service times, we ignore writebacks. Note that we use separate circuits of the same algorithm for prefetch and demand requests.

Counter name	**Purpose**
FirstAccess	Time of the first miss in that interval
LastAccess	Time of last completed miss
OutstandingMisses	Current in-flight misses
TotalMisses	Total completed misses in that interval
ElapsedCycles (intermediate)	Cycles spent on servicing *TotalMisses*
TotalElapsedCycles (at end of interval)	Total cycles spent on servicing *TotalMisses*
AvgServiceTime	Holds the avg service time

Table 4.1: Set of counters used in Estimation of Average Miss Service Time.

serviced back from the DRAM. The use of FirstAccess, LastAccess, OutstandingMisses and TotalMisses counters ensure that overlapping of misses is taken into consideration while estimating average miss service times. Precisely, the time gap (in cycles) between LastAccess and FirstAccess counters when OutstandingMisses counter is zero indicates the cycles that have elapsed while servicing TotalMisses number of misses.

At the end of an interval, average service time is estimated. Computing the total cycles elapsed during that interval depends on the value of OutstandingMisses counter, which tells the number of outstanding misses that started in that interval but, are yet to finish. If the value is not zero, our algorithm makes an approximation. It sets LastAccess counter value to the clock cycle at which the interval ends. Then, it adds OutstandingMisses counter value to TotalMisses. The difference between LastAccess and FirstAccess is added to TotalElapsedCycles counter. Finally, FirstAccess counter is set to the beginning of the next interval so that the residual cycles of the outstanding misses are accounted in the subsequent interval. On the other hand, if the value of OutstandingMisses is zero, the elapsed cycles already computed (line numbers 8 to 11 in the algorithm) gives the total elapsed cycles while servicing TotalMisses number of misses in that interval.

Selecting the Application to perform Prefetcher Aggressiveness Control

When Band-pass prefetching detects interference on demands by prefetches (using Equation 4.1), it decides to throttle-down the prefetcher of the application that issues the highest global fraction of L2 prefetch requests. This decision is inline with our observation presented in Section 4.3.2: prefetcher-caused interference (delay on demands) increases with the increase in the ratio of total prefetch to demand requests. Hence, the application with the highest L2 prefetch-fraction causes the most interference. Therefore, Low-pass component issues only 50% of the prefetch requests of this application. That is, only once in every second prefetch request is issued to the next level. Prefetchers of other applications are allowed to operate in aggressive mode. Similarly, when

Algorithm 1 Estimation of Average Miss Service Time

On a new Miss at LLC
OutstandingMisses++
FirstAccess=CurrentClock if its Reset
When a Miss is Serviced back from DRAM
−−*OutstandingMisses*
LastAccess=CurrentClock
TotalMisses++
if *OutstandingMisses* == 0 **then**
    ElapsedCycles=(*LastAccess*-*FirstAccess*)
    TotalElapsedCycles+=*ElapsedCycles*
    FirstAccess=*LastAccess*=0 //Reset
end if
At the end of an Interval
if *OutstandingMisses*≠ 0 **then**
    TotalMisses+=*OutstandingMisses*
    LastAccess= CurrentClock
    ElapsedCycles=(*LastAccess*-*FirstAccess*)
    TotalElapsedCycles+=*ElapsedCycles*
    FirstAccess=Beginning of Next Interval
else
    FirstAccess=*zero* //Reset
end if
$\text{AverageServiceTime} = \frac{\text{TotalElapsedCycles}}{\text{TotalMisses}}$
TotalMisses=*ElapsedCycles*=*TotalElapsedCycles*=0

Band-pass prefetching detects prefetcher-caused interference to be low, it allows all prefetchers to operate in aggressive mode.

Measuring Global Prefetch-fraction of Applications

The method of measuring global prefetch-fraction at the shared LLC-DRAM interface is similar to High-pass prefetching except the fact that the total requests measured by Low-pass correspond to the requests from all applications at the LLC-DRAM interface.

4.4.3 Overall Band-Pass Prefetcher

Figure 4.5 shows the logical diagram of our proposed band-pass prefetching mechanism. The high-pass and low-pass filters operate independently. Both the components are triggered at the end of interval. From experiments, we fix 1 Million LLC misses as the interval size. High-pass filter computes the local prefetch-fraction of each application at the L2-LLC inerface while the low-pass filter computes the global prefetch-fraction of each application at the LLC-DRAM interface. From the feedback collected about

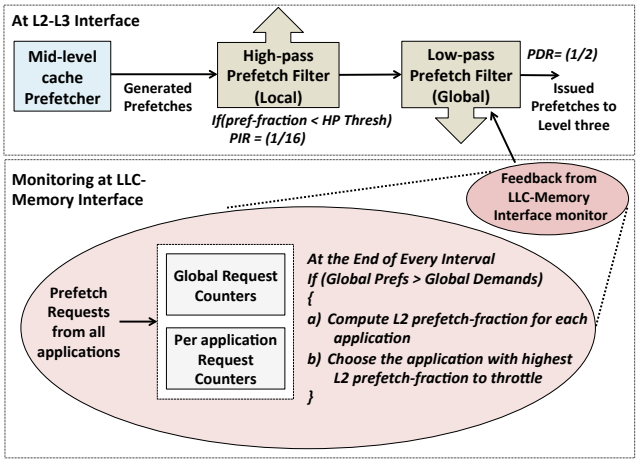


Figure 4.5: Schematic diagram of Band-pass Prefetching. PI(D)R: Prefetch Issue(Drop) Rate, pref-fraction: prefetch-fraction and HP Thresh: High-pass Threshold

the local and global prefetch-fraction, the two filters control the prefetch issue rate. Band-pass prefetching components and estimation of average memory service times, prefetch-fraction estimation do not require modification to cache tag arrays or MSHRs. All measurements lie outside the critical path.

4.5 Experimental Setup

4.5.1 Baseline System

We use cycle-accurate BADCO [VMS12] x86 CMP simulator which models 4-way OoO core with a cache hierarchy of three levels. Level 1 and Level 2 caches are private. The last level cache and the memory bandwidth are shared by all the cores. Similar to prior studies [WJM+11, SYX+15, EMLP09, PB15, Pan16], we model bank-conflicts but with fixed access latency across all banks. Cache line size is 64 bytes throughout the hierarchy and we do not enforce inclusion across cache levels. Our prefetcher model is as described in Section 4.2.1. Other system parameters are available in Table 5.1. A VPC [NLS07] based scheduler arbitrates requests from L2s to LLC. Other system parameters are available in Table 5.1.

4.5.2 Benchmarks and Workloads

We use SPEC CPU 2000, 2006 [SPE], and PARSEC [Bie11] benchmark suites totalling 34 (31+3) plus one stream benchmark. Similar to prior studies [EMLP09, PB15, Pan16], we classify benchmarks based on their IPC improvement over no prefetching when run alone (Table 5.2). We construct four types of workloads, namely mixed-type, highly prefetch-friendly, medium prefetch-friendly and prefetch-unfriendly. Table 4.4 lists each

Processor	4-way OoO, 4.8GHz (ROB,RS,LD/ST) 128, 36, 36/24
Branch predictor	TAGE, 16-entry RAS
IL1 and DL1	32KB, LRU, next-line prefetch ICache:2-way, DCache:8-way
L2(unified)	256 KB, 16-way, DRRIP 14-cycle, MSHR:32-entry
LLC (unified and shared)	16MB, 16-way,PACMAN 24-cycle, 256-entry MSHR, 128-entry WB
Memory controller (channels-rank-bank) (4-1-8) for 16-cores	FR-FCFS with prefetch prioritization[LMNP08] (TxQ,ChQ) : (128,32)
DDR3 parameters	(11-11-11), 1333 MHz IO Bus frequency : 1600MHz

Table 4.2: Baseline System Configuration.

Category	**Benchmarks**
Highly prefetch-friendly (class A) [IPC $\geq$ 10%]	apsi, cact, lbm, leslie, libq, sphn, STREAM
Medium prefetch -friendly (class B) IPC [$\geq$ 2%,<10%]	blackscholes, facesim, hmm, mcf, vpr, wup, streamcluster
Prefetch-unfriendly (class C) IPC [$\pm$ 2%]	art, astar, bzip, deal, gap, gob, gcc, gzip, milc, omn, pben, sop, twol, vort

Table 4.3: Classification of benchmarks.

workload type and its construction methodology using the benchmarks as classified in Table 5.2.

In total, we study 66 16-core multi-programmed workloads. In our experiments, we use the portion of benchmarks between 12 to 12.5 billion instructions. In that phase, the first 200 million instructions of each benchmark warm-up all the hardware structures. The next 300 million instructions are simulated. Simulations are run until all benchmarks finish 300 million instructions. If a benchmark finishes execution, it is rewind and re-executed. Statistics are collected only for the first 300 million instructions.

Type	**#Benchmarks**	**#Workloads**
Mixed	(5,5,6), (5,6,5) (6,5,5)	12 each
(Type A) Highly prefetch-friendly	(10,3,3)	20
(Type B) Medium prefetch-friendly	(3,10,3)	20
(Type C) prefetch -unfriendly	(3,3,10)	20

Table 4.4: Workload Types and their Composition.

4.6 Results and Analyses

We first present the performance results of High-pass Prefetching, our local component (at the private L2 to LLC interfaces) that throttles prefetch requests of an application based on its prefetch-fraction. Then, we present the performance results of Band-pass prefetching that consists of both High-pass and Low-pass filter components. Throughout our study, we use harmonic speedup (HS) [LGF01] since it balances both system fairness and throughput.

4.6.1 Performance of High-pass Prefetching

High-pass Prefetching dynamically computes prefetch-fraction of an application at its private L2-LLC interface to infer usefulness (accuracy) of prefetching. If the computed prefetch-fraction value is below a threshold, it throttles-down prefetch requests. Here, we study the sensitivity of prefetch-fraction threshold values on High-pass Prefetching. Table 4.5 shows the performance of High-pass Prefetching in terms of harmonic speedup across 12 High-pass threshold values (between 9% and 42% in steps of 3%). Performance is normalized to the baseline that implements aggressive prefetching without prefetcher throttling. Results are averaged (geometric-mean) across 36 mixed-type workloads. The goal of High-pass is to throttle-down useless prefetch requests and avoid interference caused due to them. When the thresholds increase from 9% to 21%, performance also increases. However, beyond threshold value of 21%, performance begins to drop. Because, as the threshold increases beyond 21%, useful prefetch requests are also throttled-down by High-pass (recall from Section 4.3.1 that prefetch-accuracy increases with increase in prefetch-fraction). Therefore, we fix High-pass threshold at 21%. Table 4.5 also shows reduction in (i) number of prefetch requests issued and (ii) bus transactions due to High-pass prefetching as compared to aggressive prefetching that uses no prefetcher throttling.

High-pass Threshold	Perf over Aggr Pref	Redn. in #Prefs	Redn. in #Bus Trans
9	1.17	1.6	0.34
12	1.28	3.2	1.06
15	1.36	3.39	1.2
18	1.35	3.56	1.22
21	**1.37**	**3.93**	**1.36**
24	1.36	4.42	1.5
27	1.31	4.61	1.54
30	1.21	5.5	1.79
33	1.00	7.58	2.07
36	0.99	14.2	3.37
39	0.98	17.28	3.94
42	0.95	32.08	5.71

Table 4.5: Performance improvement of High-pass Prefetching over Aggressive Prefetching across High-pass thresholds. All quantities are presented in percentage.

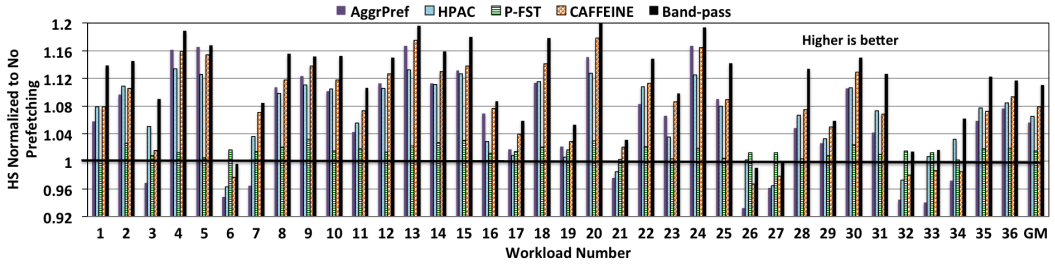


Figure 4.6: Performance of prefetcher aggressiveness control mechanisms. GM: Geometric Mean.

4.6.2 Performance of Band-pass Prefetching

We present Band-pass prefetching that consists of both High-pass and Low-pass components together since Low-pass component directly handles prefetcher-caused interference at the shared LLC-DRAM interface, and hence, it is the major performance contributor. Figure 4.6 shows the performance of Band-pass across the 36 mixed-type workloads in terms of harmonic speedup. It also shows the performance of aggressive prefetching that does not use prefetcher throttling, state-of-the-art HPAC[6], P-FST [ELMP11] and CAFFEINE. The x-axis represents workload numbers, and the y-axis shows harmonic speedup normalized to no prefetching. Over no prefetching baseline, Band-pass achieves 11.1% on average, and up to 20.47% on workload 20, while HPAC, P-FST and CAFFEINE achieve 6.4%, 1.5% and 7.7% improvement, respectively. Aggressive prefetching with no prefetcher throttling achieves 5.6%. In the following paragraphs, we give an overview of each of these mechanisms, and in Section 4.6.5 we discuss a case study to

[6]We tune the thresholds of HPAC and P-FST to suit the system configuration that we study.

understand the mechanisms in detail.

Analysis: When compared to aggressive prefetching, HPAC degrades performance on workloads that benefit from aggressive prefetching (4, 5, 13, 20 and 34). On workloads that suffer highly from aggressive prefetching (3, 6 and 7), HPAC is not able to completely mitigate prefetcher-caused interference. The use of multiple metrics (driven by their thresholds) does not reflect the actual interference in the system and causes HPAC to make incorrect throttling decisions, and makes it less effective. In contrast, Band-pass prefetching is able to retain the benefits of aggressive prefetching as well as effectively mitigate prefetcher-caused interference achieving 4.6% improvement over HPAC.

P-FST's model of determining the most-interfering application and its use of multiple metrics on top of HPAC together lead to incorrect throttling decisions, and forces prefetchers of several applications to conservative mode. Therefore, P-FST achieves low performance improvement (close to 1.5%) over no prefetching. For the same reason, on workloads where aggressive prefetching is beneficial, P-FST decreases performance. In contrast, Band-pass prefetching is able to achieve higher performance improvement compared to P-FST. On workloads where aggressive prefetching is harmful, Band-pass achieves either comparable (at most ±2% on workloads 6, 21, 27 and 32) or higher performance improvement (workloads 3 and 7). Overall, Band-pass achieves 9.6% over P-FST.

With CAFFEINE, on workloads in which aggressive prefetching is beneficial (workloads 4, 5, 13, 20 and 30), it achieves performance improvement comparable to Band-pass and aggressive prefetching mechanisms. On certain prefetch-friendly workloads (workloads 15, 24, 29, 31 and 35), Band-pass prefetching is still able to achieve higher performance over CAFFEINE thanks to its effective mechanism of detecting interference. However, on workloads that suffer highly due to prefetcher-caused interference (workloads 3,26 and 34), Band-pass outperforms CAFFEINE as CAFFEINE is not able to capture prefetcher-caused interference due to its approximate estimation of miss-penalty. Overall, Band-pass prefetching achieves 3.2% improvement over CAFFEINE.

In summary: Band-pass prefetching is able to retain the benefit of aggressive prefetching as well as effectively manage prefetcher-caused interference. However, state-of-the-art prefetcher aggressiveness control mechanisms are either conservative in cases where aggressive prefetching is actually beneficial (HPAC and P-FST), or do not completely mitigate prefetcher-caused interference (HPAC and CAFFEINE).

4.6.3 Impact on Average Miss Service Times

Band-pass prefetching uses the ratio of average miss service times of demands and prefetches as one of its throttling conditions (Equation 4.1). It attempts to decrease the total number of prefetch requests in the system as compared to demands. In doing so, it reduces the interference caused on demands by prefetches in terms of their LLC miss service times, which in turn translates to performance improvement.

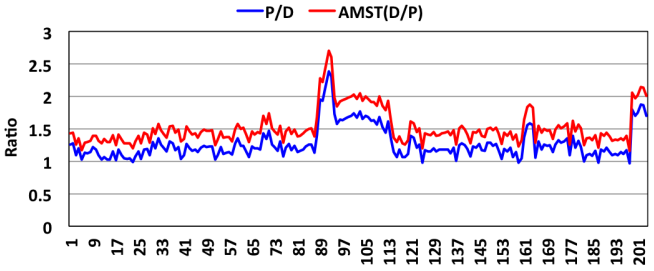


Figure 4.7: Ratio of LLC miss service times demand to prefetch and number of prefetch to demands in the system under Aggressive Prefetching

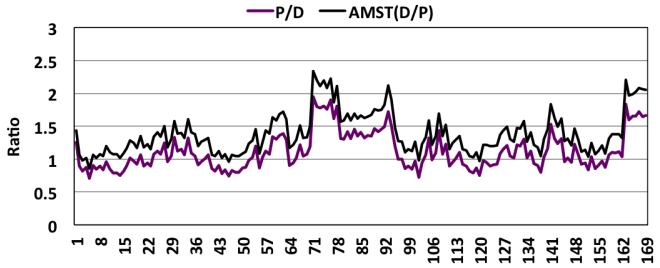


Figure 4.8: Ratio of LLC miss service times demand to prefetch and number of prefetch to demands in the system under Band-pass Prefetching

Figures 4.7 and 4.8 show the ratio of prefetch to demand requests and the ratio of average memory service times of demand to prefetches during the execution of workload 3 under aggressive prefetching and Band-pass prefetching, respectively. The x-axis represents execution time in intervals of 1 Million LLC misses, while the y-axis represents the ratio of the two quantities. As can be seen from the two figures, the ratio of AMST(D/P) is higher in Figure 4.7 as compared to Figure 4.8. The average AMST(D/P) on this workload under aggressive prefetching is 1.51, which becomes 1.35 under Band-pass prefetching. That is, Band-pass prefetching reduces the average service time on demands 10.59%. Band-pass prefetching effectively identifies interference happening due to prefetches by checking (P/D) in Equation 4.1. Overall, as compared to aggressive prefetching, Band-pass reduces the ratio of average service times of total demands to prefetch requests on average by 18% (from 2.0 to 1.64), while increasing the average service time of prefetch requests by 9.5%

4.6.4 Impact on Off-chip Bus Transactions

Figure 4.9 shows the percentage increase in bus transaction due to prefetching as compared to No Prefetching. Aggressive prefetching increases bus transactions by 14.3% while P-FST shows the least increase (only 1.3%) because of its conservative prefetcher throttling as described in Section 4.6.2. As compared to aggressive prefetching, Band-

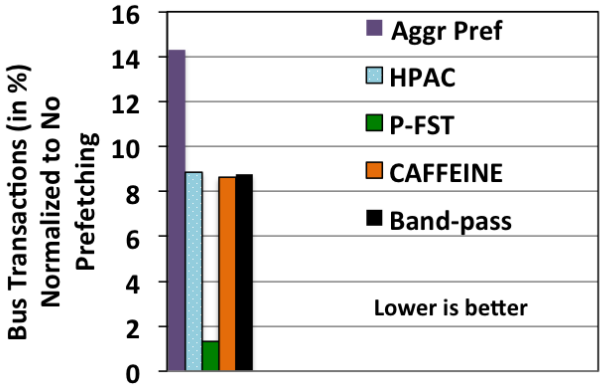


Figure 4.9: Increase in Bus Transactions as compared to No Prefetching.

pass prefetching reduces the bus transactions by 5.55% while achieving better performance of 5.17%. When compared to HPAC and CAFFEINE, Band-pass achieve performance improvement of 4.6% and 3.19%, respectively, while incurring comparable bus transactions.

4.6.5 Understanding Individual Mechanisms

In order to gain insights on the individual mechanisms, we discuss a case study of workload 3, which shows the scenario where state-of-the-art HPAC and CAFFEINE do not completely mitigate prefetcher-caused interference. Figure 4.10 shows the IPC of individual benchmarks normalized to No Prefetching.

HPAC: Under HPAC, libq slows-down by 19% as compared to aggressive prefetching from 1.38 to 1.19 (Figure 4.10). This is because, useful and timely prefetch requests of libquantm get delayed by memory requests of other applications. Its prefetch-accuracy drops to around 35% (which is below HPAC's prefetch-accuracy threshold). Hence, HPAC throttles-down libq's prefetcher[7] for successive intervals to conservative mode. Under such a scenario where prefetch-accuracy is low, FDP does not throttle-up the prefetcher as it intends to save bandwidth by throttling-down prefetchers that have low prefetch-accuracy. Therefore, its prefetcher gets stuck in conservative mode, and not able to exploit the benefit of prefetching. Altogether, HPAC does not detect the interference caused on libq and decreases its performance.

P-FST:P-FST's interference models identify benchmarks such as cactusADM, libq, apsi, deal and lbm to be interfering, and conservatively throttle-down their prefetchers on most intervals. Benchmarks such as hmmer, facesim and vpr improve on their performance while most others do not. Since it throttles-down most of its prefetchers, only

[7]HPAC also observes high value of BWNO for *libq*. Using the two metrics, HPAC's global component throttles-down its prefetcher (as mentioned in Section 4.2.2).

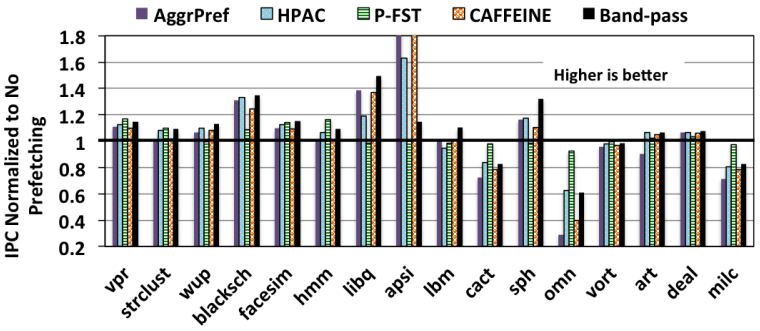


Figure 4.10: Normalized IPCs of each benchmarks of workload 3.

few benchmarks are able to exploit the benefit of prefetching. Hence, P-FST is able to achieve only marginal increase in performance as compared to no prefetching.

CAFFEINE: CAFFEINE observes positive system-wide net-utility due to prefetching on this workload. Because, CAFFEINE's approximate miss-latency model overestimates the cycles saved due to prefetching. Hence, applications with high prefetch-accuracy such as apsi(96%) and lbm (83%) bias system-wide net-utility metric in favor of prefetching. Therefore, though apsi consumes high bandwidth, CAFFEINE does not detect interference due to apsi and does not throttle-down its prefetcher on most intervals. From Figure 4.10, we observe benchmarks such as omnet, milc and cactusADM suffer slow-down due to interference.

Band-pass prefetching: Band-pass prefetching computes prefetch-fraction of applications at the shared LLC-DRAM interface to identify the most interfering application. Using prefetch-fraction, it effectively identifies apsi as most-interfering application, and throttles-down its prefetcher. Though the normalized IPC of apsi decreases from 2.05 to 1.29, Band-pass improves the IPCs of benchmarks such as libq, omnet, sphinx and lbm. Overall, Band-pass improves the performance of this workload by 13% as compared to aggressive prefetching, while HPAC and CAFFEINE improve performance by 8% and 5%, respectively.

4.6.6 Sensitivity to Workload Types

Figure 4.11 shows the performance of prefetcher aggressiveness control mechanisms across various workload types. Performance is normalized to aggressive prefetching that uses no prefetcher throttling. Table 4.4 summarizes the workload types and their composition. P-FST suffers heavy slow-down on type A and type B workloads. Hence, we ignore its results here. On type A, HPAC degrades performance close to 4% as compared to aggressive prefetching. CAFFEINE achieves marginal improvement over the baseline. Though type A workloads are highly prefetch-friendly, Band-pass is still able to effectively identify interference and improve performance close to 3.35%. When compared to HPAC, it achieves 5.91% additional performance.

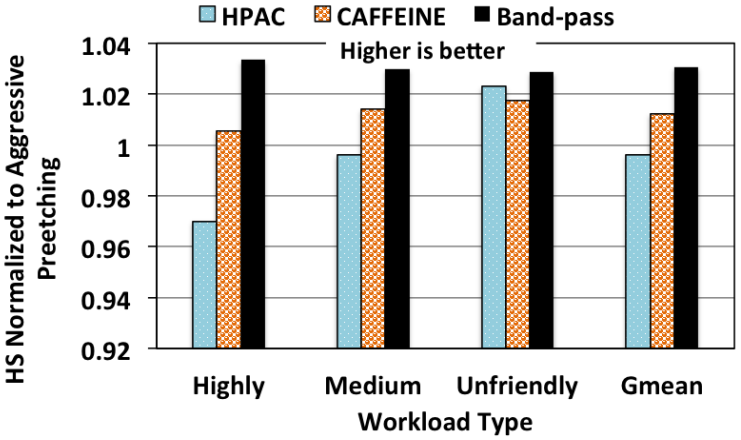


Figure 4.11: Sensitivity to Workload types.

On type B, performance of HPAC is close to that of baseline. CAFFEINE achieves close to 1.4 % while Band-pass achieves higher performance (close to 3%) than CAFFEINE. On type C workloads, all three mechanisms achieve comparable performance (within range of 0.5%). This is because the overall number of prefetches in the system as compared to demands is not high (applications in this workload category have smaller global prefetch-fraction values). Therefore, the impact of prefetching on demand is not significant on this workload category. Overall, Band-pass prefetching improves performance across different workload types, and hence, we infer our mechanism is robust.

4.6.7 Sensitivity to Design Parameters

Impact of Prefetch Drop Rate: Figure 4.14 shows the sensitivity of Band-pass prefetching to prefetch drop rates (represented as PDR) on the mixed category workloads. Recall that our mechanism throttles-down prefetch requests by not issuing (dropping) them to the next level. Increase in prefetch drop rate increases the performance up to 5.15% (PDR=1/2), beyond which it saturates. That is, dropping prefetch requests of the most-interfering application beyond 50% does not improve performance further. Therefore, we fix the prefetch drop rate at 1/2.

Impact of L1 Prefetch Requests on Prefetcher Throttling Decisions: The throttling condition in Equation 4.1 considers only L2 prefetch requests. We study the impact of including L1 Prefetch requests in the throttling decisions. Therefore, TP in TP/TD of Equation 4.1 now represents (P1+P2), where P1 and P2 reperesent total L1 and L2 prefetch requests, respectively. Figure 4.12 compares the performance of this design against the former across workload types. Including L1 prefetch requests, marginally increases the number of intervals in which TP/TD is greater than one, and hence, the number of intervals in which prefetcher aggressiveness control is applied. On workloads where aggressive prefetching is harmful, this design marginally increases the perfor-

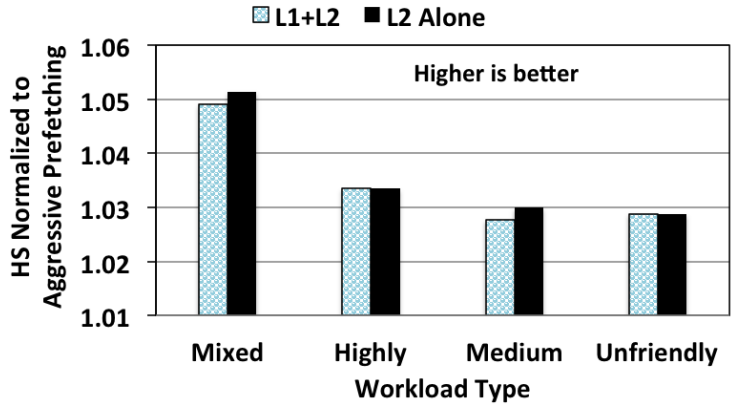


Figure 4.12: Impact of including L1 Prefetch Requests on Throttling Decisions of Equation 4.1.

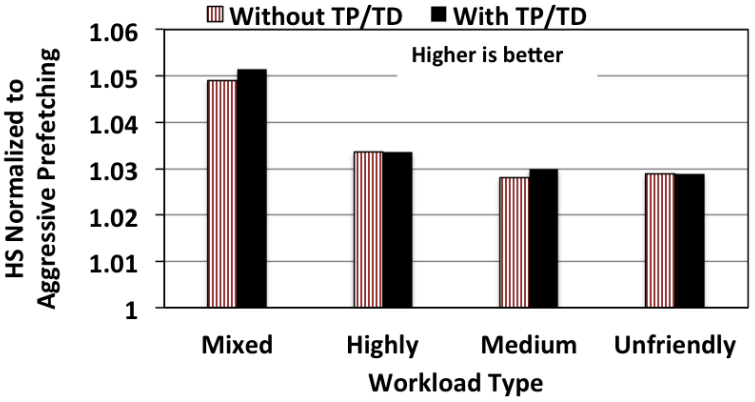


Figure 4.13: Impact of checking TP/TD ratio on Prefetcher Throttling Decisions.

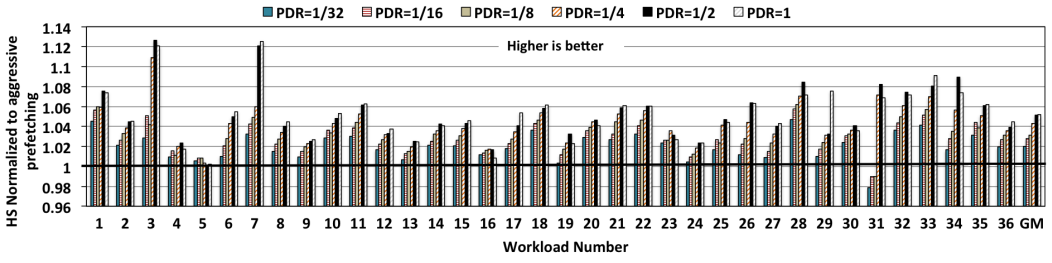


Figure 4.14: Sensitivity of Band-pass Prefetching to Prefetch Drop Rate (PDR). GM: Geometric Mean

mance. However, on workloads that benefit from aggressive prefetching, performance degrades marginally. Overall, there is only tiny performance difference between the two designs. Therefore, we conclude that L1 prefetch requests do not have significant

impact on our mechanism.

Impact of TP/TD on Prefetcher Throttling Decisions: Equation 4.1 presents the conditions under which Band-pass performs prefetcher throttling. To understand the impact of TP/TD on throttling decisions, we ignore TP/TD condition in Equation 4.1 and compare only the average miss service times of demands and prefetch requests in making throttling decisions. Figure 4.13 shows the performance of this design across workload types. We observe that comparing TP/TD marginally improves the performance on certain workloads, while having no impact on others. Though comparing TP/TD yields marginal benefit, we observe that without comparing TP/TD, prefetchers, in some cases, are conservatively throttled although interference due to prefetchers is not significant. Hence, we include TP/TD in making throttling decisions.

4.6.8 Sensitivity to AMPM Prefetcher

In this section, we evaluate Band-pass prefetching on systems that use AMPM [IIH09] as the baseline prefetching mechanism. We briefly describe below.
AMPM: AMPM uses a bit-map to encode the list of cache lines accessed in a given region of memory addresses (adapted to 4KB in our study). Each cache line can be in one of the four states: init (initial state), access (when accessed by a demand request), prefetch (when it is prefetched) or success (when the prefetched cache line receives a hit). When there is a demand access to a cache line in a region, AMPM uses the bitmap to extract the stride/offset values from the current demand access. From the prefetchable candidates, if the state of each candidate cache line is either access or success, AMPM issues prefetch requests. In this way, AMPM is able to convert most of the demand requests into prefetch requests.
Figure 4.15 shows the performance of Band-pass prefetching across the 36 mixed-type workloads in terms of Harmonic Speedup (HS). Over no prefetching baseline, Band-pass achieves the highest average performance of 12.9%, while HPAC, P-FST, and CAFFEINE achieve 7.8%, 10.8%, and 11.8%, respectively. Interestingly, P-FST achieves higher performance as compared to HPAC. This is because of AMPM's prefetching methodology and P-FST's interference model. P-FST accounts interference caused by a prefetch or a demand request only on the other core's demand requests, and not on the prefetch requests. Therefore, in cases where the demand requests of most applications get converted to prefetch requests (due to AMPM), P-FST does not account interference caused on prefetch requests. Hence, on most intervals, unfairness estimate on individual applications is lower than the unfairness threshold, and P-FST allows the prefetchers to run aggressively. On the other hand, HPAC, as before, performs prefetcher throttling based on threshold values of metrics, which is not effective.

CAFFEINE and Band-pass outperform each other on most workloads because of the throttling methodology of the respective mechanisms. On certain prefetch-friendly workloads in which aggressive prefetching is beneficial (workloads 4, 5, 10, 12, 14, 16, 17, 19), CAFFEINE and Band-pass achieve comparable performance. On few prefetch-friendly workloads like 8, 13, 20, and 24, CAFFEINE provides higher performance than Band-pass, while on workloads 2, 15, 25, 30, and 35, Band-pass is able to

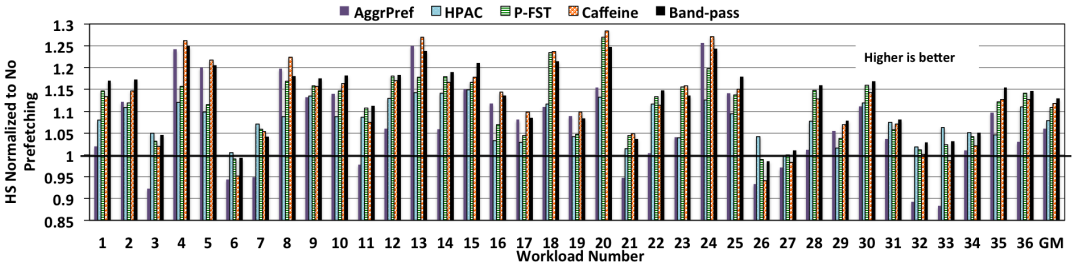


Figure 4.15: Performance of prefetcher aggressiveness control mechanisms. GM: Geometric Mean.

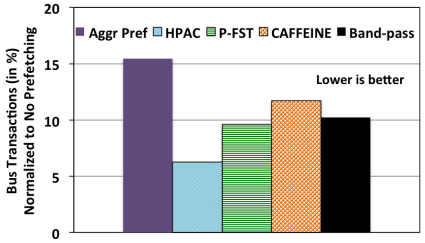


Figure 4.16: Increase in Bus Transactions as compared to No Prefetching.

achieve higher performance. Overall, Band-pass achieves higher performance by 1.1% over CAFFEINE.

4.6.8.1 Impact on Off-chip Bus Transactions:

Figure 4.16 shows the percentage increase in bus transactions due to prefetching as compared to No Prefetching. Aggressive prefetching increases bus transactions by 15.4%, while HPAC shows the least increase (6.27%) because of its conservative throttling. When compared to aggressive prefetching, P-FST and CAFFEINE, Band-pass achieves higher performance of 6.9%, 4.9% and 1.1%, respectively, while incurring 5.2% (fewer), 1.5% (fewer) and 0.6% (higher) bus transactions.

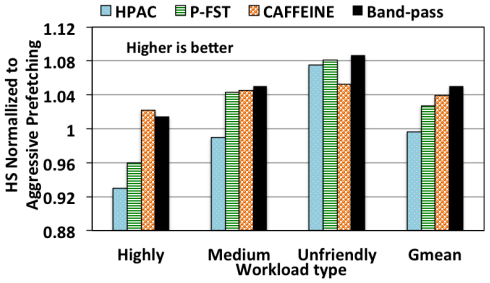


Figure 4.17: Sensitivity to Workload types.

4.6.8.2 Sensitivity to Workload Types:

Figure 4.17 shows the performance of the prefetcher aggressiveness control mechanisms across various workload types. On Type-A workloads, which are highly prefetch-friendly, both HPAC and P-FST slow-down performance. However, CAFFEINE and Band-pass achieve higher and comparable (<1%)performance. On type B workloads, performance of HPAC is close to that of baseline, while the other three provide comparable performance close to 4%. On type C workloads, CAFFEINE achieves only 4% improvement over the baseline while the other three mechanisms provide comparable performance improvement (8%). Overall, Band-pass provides consistent performance improvement across different workload types, while P-FST and HPAC show slow-down on prefetch-friendly and medium prefetch-friendly workloads. Similarly, CAFFEINE is not able to mitigate interference completely on prefetch-unfriendly workloads. Therefore, we infer that Band-pass prefetching is robust across different workload types.

4.6.9 Using prefetcher-accuracy to control aggressiveness

We performed experiments where we apply throttling on the application with least prefetch-accuracy. However, there was no improvement over aggressive prefetching, because low accuracy does not correlate/imply high interference. We further experimented with a combination of prefetch-accuracy and prefetch-fraction to throttle the prefetcher that issues the highest fraction of in-accurate prefetch requests. That is, we measure inaccuracy (1-accuracy) multiplied by prefetch-fraction, which gives the fraction of inaccurate prefetches/application. Again, there was only marginal improvement over aggressive prefetching, because in most cases this metric does not capture the most interfering application. For ex, on Workload 3 on Figure 7, apsi, which has very high-accuracy but causes high interference, will have very low inaccurate fraction as compared to others. Hence, we conclude using prefetch-accuracy for interference is not effective.

4.6.10 Hardware Overhead

High-pass and Low-pass prefetching require 53 bits and 37 bits per application, respectively. The first part of Table 4.6 shows the counters that are common to both the components. High-pass prefetching requires TotalCounter per application. However, Low-pass prefetching requires only one TotalCounter (32-bit in size) is required since it measures global prefetch-fraction of applications, only one TotalCounter. Hence, we save 16-bit per application for the Low-pass component. To measure interval size in terms of LLC misses, we use a 20-bit counter. Estimation of average miss service times of prefetch and demand requests requires seven counters each (Table 4.1). Each counter is 32-bit in size and the total cost amounts to 56 bytes of storage. For a 16-core system, hardware overhead is only 239 bytes, while HPAC, P-FST and CAFFEINE require

about 208KB, 228.5KB and 204KB, respectively. Note that Band-pass prefetching does not require any modification to cache tag arrays or MSHR structures.

Counter	**Purpose**	**Size**
L2PrefCounter	Records L2 prefetches	16-bit
Pref-fraction	Stores prefetch-fraction	16-bit
Drop bit	To drop or Not to drop	1-bit
InsertCounter	For probabilisitic Insertions	4-bit
TotalCounter (High-pass)	Records total requests of an appplication	16-bit
TotalCounter (Low-pass)	Records total requests at shared LLC-DRAM Interface	32-bit

Table 4.6: Hardware Overhead of Band-pass Prefetching.

4.7 Conclusion

In this chapter, we discussed our contribution, Band-pass prefetching, a simple and effective mechanism to manage prefetching in multicore systems. Our solution builds on the observations of strong correlation between (i) prefetch-fraction and prefetch-accuracy and (ii) ratio of the average miss service times of demand to prefetch requests and the ratio of prefetch to demand requests in the system. The first observation infers the usefulness (in terms of prefetch-accuracy) of prefetching while the second observation infers the prefetcher-caused interference on demand requests.

Our mechanism consists of two prefetch filter components: High-pass, which is present at the private L2-L3 and Low-pass component is present at the shared LLC-DRAM interface. The two components independently compute prefetch-fraction of applications at the private L2-LLC and shared LLC-DRAM interfaces. Together, the two components control the flow of prefetch requests between a range of prefetch-to-demand ratios. Experimental results show that Band-pass prefetching achieves 11.1% improvement over the baseline that implements no prefetching. We further experiment our mechanism on systems that uses AMPM prefetcher and observe Band-pass to show similar performance trends : 12.9% improvement over the baseline that implements No Prefetching. Experimental studies show Band-pass is effective in mitigating interference caused by prefetchers, and is robust across workload types.

Chapter 5

Reuse-aware Prefetch Request Management

In this chapter, we present the discussion on handling prefetch requests at the shared last level cache.

5.1 Introduction

Prefetching is a latency hiding mechanism that attempts to reduce the time spent by an application on an off-chip memory access. A prefetched data is placed in the on-chip cache as with the data fetched by the demand accesses. Sharing the same cache space as the demand accesses introduces interference. Prefetched cache blocks may pollute the cache by evicting a more useful demand fetched data. To address pollution, some studies [LYL87, Jou90] include a separate prefetch buffer. However, storing the prefetched data directly on the on-chip cache provides two advantages. First, it simplifies coherence operations by avoiding the necessity to access the prefetch buffer and second, avoiding to invest in the design of a prefetch buffer. Srinath et. al. [SMKP07] identify that for single-core system with 1MB cache, a minimum prefetch buffer size of 32KB is needed to mitigate pollution as well as to harness the usefulness of prefetching. For a large scale multi-core processor, it would not be an exaggeration to say that the size of such prefetch buffer(s) would need to be 1MB or more, which is a large overhead in terms of storage cost. Therefore, a desirable solution is to efficiently manage the on-chip cache in the presence of prefetching.

Along this direction, prior studies such as FDP [SMKP07], PACMAN [WJM$^+$11] and ICP [SYX$^+$15] have proposed to alter the insertion and promotion priorities of prefetched caches lines. Their policy for predicting the insertion and promotion policies of prefetched cache lines is based on the observation of reuse behavior of prefetched cache blocks at the last level cache in a particular context: a single or small-scale multi-cores consisting of small 1MB or 2MB last level cache sizes. However, in the context of large-scale multi-cores (16-core and above) last level cache size is higher of the order 16MB. Under the context of larger last level caches, we observe that the reuse behavior

of prefetched cache blocks becomes different, and this implicit assumption breaks. In particular, we observe that treating the prefetch and demand requests alike during cache insertion and promotion operations is more beneficial than treating prefetched cache blocks with low priority.

5.2 Background

To handle prefetch requests at the last level cache, prior mechanisms like FDP, PACMAN have proposed to alter the insertion priority of prefetched cache lines. FDP proposes to insert the prefetched cache block at different locations of the LRU chain, instead of MRU insertion, depending on the degree of prefetcher-caused pollution. The idea is to allow a prefetched cache block to stay in the cache for a shorter duration of time, if it is found to be polluting. While this mechanism efficiently manages the insertion priority of prefetch requests, it works best on caches where the prefetcher resides at the last level. In state-of-the-art high performance processors [Cor, arc], cache-hierarchy consists of three levels, where the second level is private and the last level is shared, and the prefetcher sits beside the L2 cache [Cor, arc]. However, managing prefetch requests in the context of last level cache is different.

Wu et. al. [WJM+11] observe that the last level cache sees a filtered view of the prefetched cache block accesses. That is, when prefetched cache blocks are inserted at both private L2 and LLC [1], most of the accesses to the prefetched cache block are serviced by the level two cache. Only few accesses are referred at the last level cache. Therefore, the use characteristics of prefetched cache block at the last level cache is different from what FDP observes. In particular, PACMAN observes that majority of the prefetched cache blocks are never used or single used. Based on this observation, PACMAN proposes to insert the prefetched cache blocks with least-priority. In the context of RRIP chain, prefetched cache blocks are inserted with distant reuse priority (RRIP 3). Similarly, when a prefetch request hits at the last level cache, the cache lines are not promoted to higher priority as opposed to the traditional promotion policy used by replacement policies. By inserting the prefetched cache block at the distant reuse position and not promoting the cache lines that hit for prefetch requests, PACMAN attempts to minimize the life duration of prefetched cache blocks at the LLC. By doing so, PACMAN prioritizes the demands ahead of prefetch requests at the LLC.

Seshadri et. al. [SYX+15] observe a similar pattern on use characteristics of prefetch requests at the last level cache managed by LRU. Since LRU inserts the cache lines with MRU priority, prefetch requests, which are mostly single or no use cache blocks, may stay at the cache for longer duration of time and pollute demand requests. Therefore, inserting them with LRU priority avoids this problem. However, they observe that in most cases, least priority insertions forces the prefetched cache blocks to be evicted much before they are used by the demand accesses, which renders prefetching less useful. To avoid such a situation, ICP allows the insertion of prefetched cache block at MRU position as with demand fetched cache block. However, the difference is that the cache

[1] Prefetched cache blocks are inserted at both levels because the prefetcher sits at L2.

block is demoted as soon as a demand request accesses the cache block. By doing so, ICP preserves the usefulness of prefetched cache blocks as well as minimizing its duration.

On inserting a prefetch request, its insertion priority is determined by prediction. PACMAN always (statically) predicts that prefetched cache blocks are single or no-use cache lines. Therefore, it inserts with least, that is, distant reuse priority. FDP, on the other hand, makes dynamic prediction of insertion priority based on prefetch-accuracy estimated at run-time. Seshadri et. al. observe that the dynamic prediction mechanism used by FDP is not always efficient. In particular, FDP inserts the prefetched cache blocks with LRU priority when prefetch-accuracy is low [2]. As discussed in the previous paragraph, inserting prefetch requests at the least priority evicts the prefetched block before it could be used. Therefore, prefetched cache blocks inserted with least priority will result in fewer hits on the prefetched data and only result in low-accuracy. This in turn causes the prefetcher of FDP to issue fewer prefetches. As a result, an accurate prefetcher could be falsely recognized as an inaccurate prefetcher.

In order to avoid this anomalous situation, ICP proposes a mechanism that augments the cache with a filter that holds the evicted prefetched cache block. When there is a demand miss and if it finds the missing cache block (address) in the filter, it is the case where an accurate prefetch block is falsely predicted as low/inaccurate. Using this feedback to correctly estimate prefetch-accuracy, ICP utilizes the performance of prefetching.

In the following section, we discuss prefetch-use characteristics in the context of large scale multi-core processors where the last level cache is much larger in size as compared to small scale multi-cores. Before that, we describe our experimental set-up upon which our observations are based.

5.3 Experimental Setup

5.3.1 Baseline System

We use cycle-accurate BADCO [VMS12] x86 CMP simulator which models 4-way OoO core with a cache hierarchy of three levels. Level 1 and Level 2 caches are private. The last level cache and the memory bandwidth are shared by all the cores. TA-DRRIP [JTSE10] is the baseline cache replacement policy. Under this algorithm, both demand and prefetched cache blocks are assigned the same priority on insertion and promotion operations. Similar to prior studies [WJM$^+$11, SYX$^+$15, EMLP09, PB15, Pan16], we model bank-conflicts but with fixed access latency across all banks. Cache line size is 64 bytes throughout the hierarchy and we do not enforce inclusion across cache levels. Our prefetcher model is as described in Section 4.2.1. Other system parameters are available in Table 5.1.

[2]Here, insertions performed by FDP are predictions because the prefetch-accuracy used in one interval is used for insertions in the next interval.

Processor	4-way OoO, 4.8GHz (ROB,RS,LD/ST) 128, 36, 36/24
Branch predictor	TAGE, 16-entry RAS
IL1 and DL1	32KB, LRU, next-line prefetch ICache:2-way, DCache:8-way
L2(unified)	256 KB, 16-way, DRRIP 14-cycle, MSHR:32-entry
LLC (unified and shared)	16MB, 16-way,PACMAN 24-cycle, 256-entry MSHR, 128-entry WB
Memory controller (channels-rank-bank) (4-1-8) for 16-cores	FR-FCFS with prefetch prioritization[LMNP08] (TxQ,ChQ) : (128,32)
DDR3 parameters	(11-11-11), 1333 MHz IO Bus frequency : 1600MHz

Table 5.1: Baseline System Configuration.

Category	**Benchmarks**
Highly prefetch-friendly (class A) [IPC $\geq$ 10%]	apsi, cact, lbm, leslie, libq, sphn, STREAM
Medium prefetch -friendly (class B) IPC [$\geq$ 2%,<10%]	blackscholes, facesim, hmm, mcf, vpr, wup, streamcluster
Prefetch-unfriendly (class C) IPC [$\pm$ 2%]	art, astar, bzip, deal, gap, gob, gcc, gzip, milc, omn, pben, sop, twol, vort

Table 5.2: Classification of benchmarks.

5.3.2 Benchmarks and Workloads

We use SPEC CPU 2000, 2006 [SPE], and PARSEC [Bie11] benchmark suites totaling 34 (31+3) plus one stream benchmark. Similar to prior studies [EMLP09, PB15, Pan16], we classify benchmarks based on their IPC improvement over no prefetching when run alone (Table 5.2). We construct workloads such that each workload composes applications from all the three categories. In total, we study thirty six workloads in which the first twelve workloads consists of 5,5,6, second set of twelve workloads consists of 5,6,5 and the third set of twelve workload consists of 6,5,5 benchmarks from the three categories, respectively. Tables 5.3 and 5.4 list the benchmarks under each workload.

In our experiments, we forward the first 12 billion instructions, and experiment between 12 billion and 12.5 billion instructions. In that 500 million phase, first 200 million instructions warm-up all the hardware structures. The next 300 million instructions are

simulated. In our experiments, simulation begins only when all the sixteen benchmarks (of the 16-core workload) finish their warm-up phase. The simulation ends only when all the benchmarks finish executing their 300 million instructions. If a benchmark finishes execution, it is rewind and re-executed. Statistics are collected only for the first 300 million instructions.

5.4 Motivational Observations

Like PACMAN and ICP, we also observe similar use behavior for prefetched cache lines at the last level cache of size 1MB. Figure 5.1 shows the use distribution of prefetched cache blocks at the last level cache which is 1MB in size. As we observe, many applications have prefetched cache blocks that are either single use or no-use cache lines. Therefore, the mechanisms proposed by FDP, PACMAN and ICP to manage prefetch requests in the context of 1MB caches hold good. However, these mechanisms do not scale with larger caches. Because, with larger caches, the (re)use characteristics of prefetched cache blocks change since the larger cache can hold larger amount of data of an application.

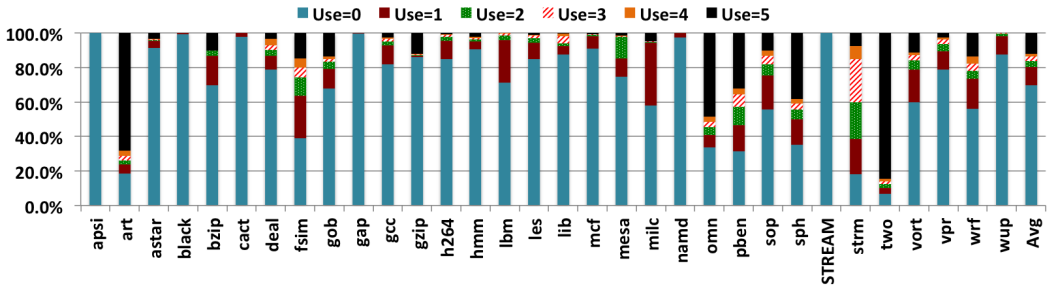


Figure 5.1: Use distribution of prefetched cache block of a 1MB last level cache.

In order to understand this behavior, lets look in detail into these figures. As we observe from Figure 5.1, 70% of the prefetched cache blocks are not used at all, while approximately 10% of the prefetched cache blocks are used once. From Figure

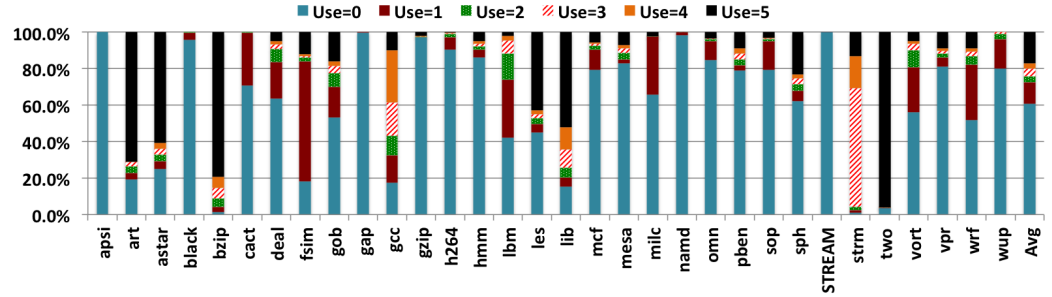


Figure 5.2: Use distribution of prefetched cache block of a 16MB last level cache.

Workloads	**Benchmarks**
WL 1	vpr, streamcluster, wup, mcf, hmm, blackscholes, apsi, libq, sphn, cact, leslie, gzip, gob, gap, art, milc
WL 2	vpr, streamcluster, wup, mcf, hmm, blackscholes, STREAM, lbm, apsi, sphn, leslie, mesa, vort, pben, astar, wrf
WL 3	vpr, streamcluster, wup, facesim, hmm, blackscholes, libq, apsi, lbm, cact, sphn, omnet, vort, art, deal, milc
WL 4	vpr, streamcluster, wup, mcf, hmm, blackscholes, libq, lbm, cact, les, apsi, vort, gap, gcc, bzip, sop
WL 5	facesim, streamcluster, wup, mcf, hmm, blackscholes, les, lbm, libq, sphn, cact, gap, bzip, astar, milc, pben
WL 6	facesim, streamcluster, wup, vpr, hmm, blackscholes, les, sphn, apsi, lbm, STREAM, bzip, milc, gap, vort, omnet
WL 7	facesim, streamcluster, wup, vpr, hmm, blackscholes, les, cact, STREAM, libq, apsi, gcc, pben, vort, omnet, art
WL 8	facesim, streamcluster, wup, mcf, hmm, blackscholes, les, libq, STREAM, sphn, apsi, vort, gzip, sop, bzip, wrf
WL 9	facesim, streamcluster, mcf, hmm, vpr, wup, les, apsi, lbm, cact, STREAM, pben, wrf, deal, bzip, mesa
WL 10	facesim, streamcluster, mcf, hmm, vpr, wup, lbm, libq, cact, apsi, les, vort, gzip, mesa, milc, gob
WL 11	facesim, streamcluster, blackscholes, hmm, vpr, wup, lbm, apsi, sphn, cact, STREAM, milc, pben, art, wrf, astar
WL 12	mcf, streamcluster, blackscholes, hmm, vpr, wup, lbm, STREAM, apsi, sphn, cact, gob, gcc, art, wrf, bzip
WL 13	mcf, streamcluster, blackscholes, vpr, facesim, libq, les, apsi, cact, STREAM, lbm, bzip, gcc, pben, sop, astar
WL 14	streamcluster, wup, mcf, facesim, vpr, apsi, les, sphn, STREAM, lbm, cact, art, gcc, bzip, sop, pben
WL 15	facesim, blackscholes, wup, vpr, mcf, apsi, les, sphn, lib, lbm, cact, astar, mesa, gob, gcc, pben
WL 16	facesim, blackscholes, wup, vpr, mcf, apsi, les, cact, libq, STREAM, sphn, lbm, gob, astar, omn, bzip, sop
WL 17	hmm, wup, blackscholes, facesim, mcf, apsi, les, lib, STREAM, lbm, cact, pben, gcc, omn, gap, astar
WL 18	hmm, wup, blackscholes, vpr, mcf, apsi, les, lib, STREAM, sphn, cact, sop, pben, art, gzip, gcc

Table 5.3: Classification of benchmarks(part1).

5.2, we observe that the percentage of zero use prefetched reduces from 70% to 60%. Applications like astar, bzip, cactusADM, deal, facesim, lbm, leslie, libquantum, gcc,

Workloads	**Benchmarks**
WL 19	hmm, wup, blackscholes, vpr, mcf, apsi, les, lib, STREAM, sphn, cact, gap, pben, astar, omnet, bzip
WL 20	hmm, streamcluster, vpr, facesim, mcf, apsi, les, lib, STREAM, lbm, cact, gap, pben, deal, sop, art
WL 21	wup, streamcluster, hmm, facesim, mcf, sphn, les, lib, STREAM, lbm, cact, gcc, bzip, milc, art, omnet
WL 22	hmm, wup, facesim, blackscholes, mcf, sphn, les, apsi, STREAM, lbm, cact, art, sop, milc, gap, astar
WL 23	facesim, blackscholes, vpr, mcf, hmm, sphn, les, STREAM, libq, lbm, cact, art, sop, vort, omnet, bzip
WL 24	streamcluster, hmm, blackscholes, facesim, mcf, sphn, les, STREAM, libq, apsi, cact, gob, gcc, pben, gzip, bzip
WL 25	blackscholes, mcf, wup, facesim, streamcluster,libq, cact,sphn, apsi, lbm, wrf, deal, milc, astar, mesa, sop
WL 26	wup, hmm, blackscholes, mcf, facesim, cact, STREAM, lbm, apsi, sphn, vort, gcc, omnet, astar, gzip, milc
WL 27	streamcluster, hmm, facesim, vpr, mcf, apsi, STREAM, les, sphn, lbm, omnet, bzip, vort, astar, sop, pben
WL 28	streamcluster, mcf, hmm, wup, vpr, sphn, apsi, les, libq, cact, gob, mesa, gcc, milc, wrf, art
WL 29	blackscholes, hmm, vpr, facesim, mcf, sphn, libq, lbm, cact, STREAM, deal, vort, bzip, gob, milc
WL 30	hmm, wup, facesim, mcf, vpr, STREAM, les, apsi, cact, lbm, astar, gzip, mesa, gob, pben, deal
WL 31	streamcluster, facesim, vpr, wupwise, hmm, libq, apsi, les, lbm, cact, oment, mesa, gob, astar, gcc, deal
WL 32	vpr, mcf, facesim, blackscholes, streamcluster, lbm, sphn, apsi, les, STREAM, art, omnet, gcc, gob, astar, vort
WL 33	vpr, hmm, facesim, streamcluster, mcf, les, STREAM, sphn, apsi, cact, gcc, art, omnet, sop, gob, wrf
WL 34	mcf, vpr, wup, facesim, streamcluster, cact, libq, sphn, apsi, lbm, wrf, bzip, omnet, milc, gob, gzip
WL 35	mcf, streamcluster, facesim, vpr, wup, apsi, libq, sphn, lbm, les, deal, wrf, gob, sop, milc, vort
WL 36	streamcluster, vpr, hmm, wup, facesim, apsi, cact, STREAM, sphn, les, gzip, deal, astar, bzip, mesa, art

Table 5.4: Classification of benchmarks(part2).

and streamcluster show large percentage of reduction in zero use prefetched cache blocks. However, applications like apsi, blackscholes, mcf, milc, and STREAM benchmarks do not show change in their prefetch use characteristics. Remaining applications show

marginal change in their prefetch use characteristics. A larger cache is able to hold more cache blocks such that when there is a reuse for a cache block, the large cache is able to retain it. However, cache blocks are evicted in the case of smaller cache. This observation holds good for applications whose working-set does not fit within a smaller 1MB cache, however working-set that fits within a large 16MB cache. Therefore, for some applications treating the prefetched cache block as zero or single use cache block could be ineffective. Further, from the two figures, we also observe that the fraction of more than twice used blocks increase from by 45% approximately, from 20% to 29%. The increase in fraction of twice used blocks suggests that several prefetched cache blocks are accessed more than once while resident in cache. In essence, discrepancy in use behaviors of prefetched cache blocks at the last level cache calls for a reuse aware mechanism to manage prefetch requests at the last level cache. Using a static mechanism that implicitly assumes zero or single use characteristics of prefetched cache blocks becomes ineffective.

5.5 Reuse-aware Prefetch Management

Prior works on prefetch request management at the last level cache such as PACMAN and ICP control the lifetime of prefetch requests by altering the insertion and cache promotion policies. Such a rigid approach accelerates the eviction of not only zero or single use prefetched cache blocks, but also multi-use (more than single use) prefetched cache blocks. This is evident from the performance of PACMAN and ICP mechanism as compared to a cache management policy that treats both prefetch and demand requests the same, which is represented as (EquallyManaged) EM in this figure. Recall that under EquallyManaged algorithm, the baseline TA-DRRIP policy applies same priorities on prefetched cache blocks as it does on demand fetched cache blocks. Figure 5.3 shows the performance of the state-of-the-art PACMAN and ICP mechanisms along with EquallyManaged mechanism that treats the demand and prefetch requests alike in cache insertion and promotion policies. Performance is normalized to no-prefetching. As we observe from Figure 5.3, on most workloads, EquallyManaged mechanism (represented as EM) provides higher performance as compared to PACMAN and ICP. On the geometric mean of average, PACMAN and ICP provide 2.7% and 4.5%, respectively, while EquallyManaged policy achieves 7.3%.

5.5.1 Understanding the Prefetch Request Management Mechanisms

In order to understand the three individual mechanisms, we study workload 5. Workload 5 consists of applications wupwise, streamcluster, mcf, hmmer, facesim, blackscholes, leslie, lbm, libquantum, sphinx, cactusADM, gap, bzip, astar, milc and perlbench, which all have different use charactersistics of their prefetched cache blocks when residing at the last level cache. As we observe from Figure 5.4, lbm, leslie and libquantum show higher performance under EquallyManaged mechanism as compared to PACMAN and ICP. The IPC numbers are normalized to no-prefetching. Under PACMAN and ICP,

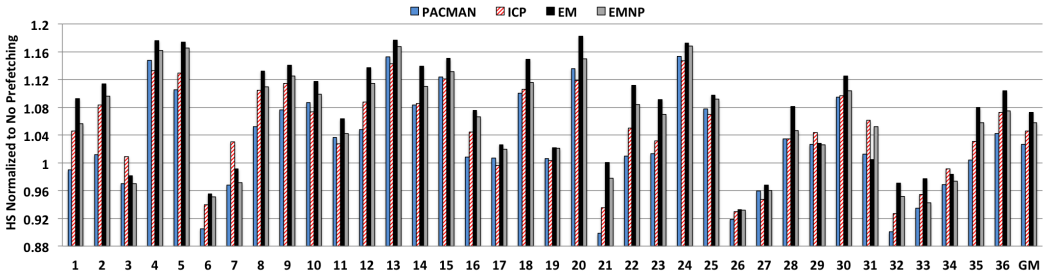


Figure 5.3: Performance comparison of state-of-the-art prefetch management mechanisms against an EquallyManaged mechanism.

leslie achieves 49% and 57% IPC improvement over no-prefetching, respectively. However, EquallyManaged mechanism achieves 71% IPC improvement over no-prefetching, which is 14.76% and 8.9% higher than what PACMAN and ICP achieve, respectively. Similarly, EquallyManaged mechanism achieves 16% and 13.6% higher performance on libquantum than what PACMAN and ICP achieve, respectively. This trend is similar for lbm as well.

Recalling from Figures 5.1 and 5.2, these applications show different use behavior of their prefetched cache blocks on a 16MB cache as compared to on a 1MB cache. Using rigid insertion and promotion policies do not allow these applications to exploit the reuse of their prefetched cache block and further enhance performance. EquallyManaged policy which inserts and promotes the prefetched cache lines on par with demand fetched cache blocks, retains the prefetched cache blocks of these applications for a longer duration of time and achieves higher hits (reducing off-chip memory accesses) and improve performance.

Since PACMAN inserts the prefetched cache blocks with distant (RRPV 3: least) reuse priority, it gives less performance improvement over no-prefetching than ICP. Most of those applications cache blocks are evicted before their use, rendering prefetching for those applications less effective. On the other hand, ICP inserts the prefetched cache lines of all applications with intermediate (RRPV 2) priority which helps to retain them longer and fetch hits. However, on applications which have either streaming or predominantly zero use behaviors, both PACMAN and ICP provide comparable performance improvement ((±2%) or no-change) over no-prefetching. Examples of such applications include wupwise, gap, blackscholes, perlbench, and mcf. On other applications which exhibit quicker reuses on their prefetched cache blocks, PACMAN is able to retain the prefetched cache lines. Therefore, all three mechanisms show comparable improvement. Applications like bzip, streamcluster, and facesim belong to this category. Under EquallyManaged mechanism, milc achieves moderately better performance (7.7% and 5%) over PACMAN and ICP because of the overall extra misses these mechanisms incur on the other applications. milc is off-chip memory access sensitive which makes it to gain better under EquallyManaged mechanism.

In order to understand the significance of promoting the prefetched cache lines on hits, Figure 5.3 also shows the performance of modified EquallyManaged mechanism

(represented EMNP) where the promotion policy for prefetched cache blocks is altered. Prefetched cache blocks are not promoted on hits, while their insertion priorities are treated on par with demand requests. As we observe from the figure, similar to EquallyManaged, modified EquallyManaged policy also achieves higher performance on most workloads. On geometric mean of average, modified EquallyManaged achieves higher performance than PACMAN and ICP, but less performance than EquallyManaged. Because modified EquallyManaged still inserts the prefetched cache blocks with the same priority as demands of its applications, most applications see more hits on the prefetched cache blocks. However, not promoting the prefetched cache lines on hits, incurs additional misses on subsequent accesses. However, not promoting the prefetched cache lines under does not incur larger penalty on lbm and libquantum because the subsequent miss on the evicted cache block is a demand access which forces the replacement policy to promote on hits. Interestingly, with leslie, there is higher penalty of not promoting the prefetched cache blocks. Because, in the case of leslie, the subsequent miss(es) on the evicted cache blocks are still prefetch requests. Therefore, the modified EquallyManaged (EMNP) mechanism keeps not promoting the prefetched cache line on hits, which operation is similar to ICP. Hence, EMNP and ICP achieve comparable performance on leslie. With these discussions, we observe promotion of prefetched cache lines which have higher number of reuses (multiple uses) is beneficial.

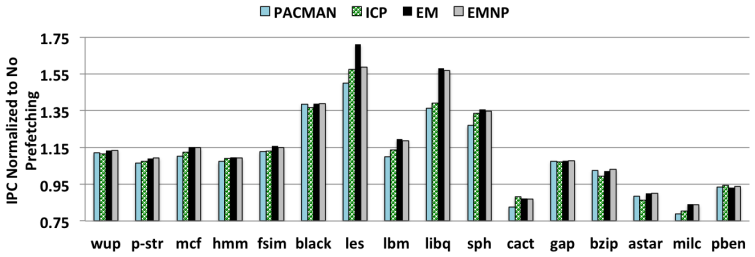


Figure 5.4: Analyzing workload number 5.

5.6 Enhancing the State-of-the-art Mechanisms

In the previous section, we discussed the significance of treating the prefetched cache lines of certain applications on par with demand requests. In this section, we use this observation to enhance the performance of the two state-of- the-art PACMAN and ICP mechanisms. In particular, we simply alter their promotion policy of prefetch requests: we treat the promotion of prefetch requests on par with demand requests. We do not alter their insertion policies. Figure 5.5 shows the Harmonic Speedup of performance of our enhancements on these two mechanisms. As before, performance is normalized to no-prefetching. The enhanced policies are suffixed with _prom.

From Figure 5.5, we observe that altering the promotion policies of the two mechanisms significantly enhances the performance on most workloads. Altering the promotion helps ICP more than PACMAN. Because, with ICP the prefetched cache lines

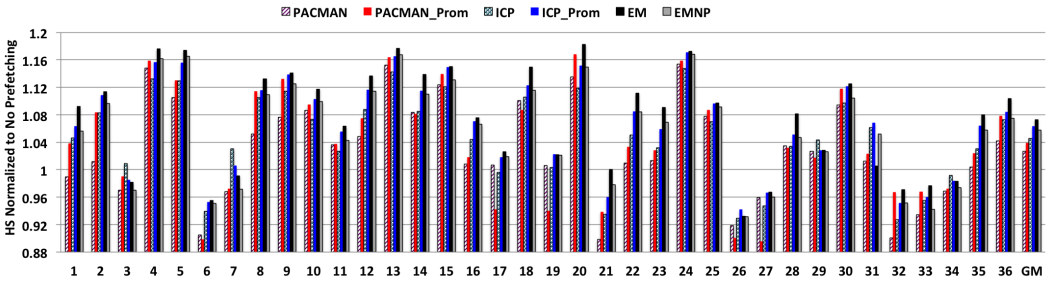


Figure 5.5: Performance of the enhanced state-of-the-art prefetch management mechanisms

are retained longer enough to observe first hit on their prefetched cache blocks which are then promoted by the enhanced promotion policy, which in turn helps to get hits on the subsequent accesses of the prefetched cache lines. However, PACMAN suffers from distant reuse (least priority) insertions which do not provide sufficient time for the prefetched cache block to stay longer before seeing a hit. Hence, promotion enhancement improves the performance of PACMAN by additional 1.1% (2.7% to 3.8%). However, enhanced promotion improves ICP by additional 2.1% (from 5.2% to 7.3%). In particular, ICP_prom marginally outperforms the modified EquallyManaged (EMNP) mechanism.

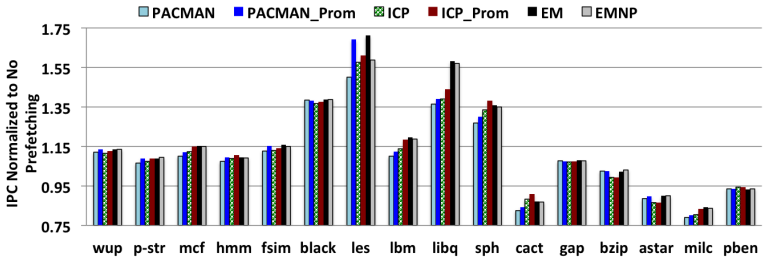


Figure 5.6: Analyzing workload number 5 under enhanced policies.

Figure 5.6 shows the IPC of the individual benchmarks of workload 5 normalized to no prefetching. As we observe the figure, enhancing the promotion policy marginally improves the performance of libquantum and lbm. Interestingly, the performance improvement on leslie is significantly high and comparable to EquallyManaged. The reason is similar to the previous discussion of modified EquallyManaged. In the case of modified EquallyManaged (EMNP), the subsequent access to the prefetched cache blocks were only prefetch requests, and not demand accesses. Since in the enhanced version, the immediate prefetch request access that follows the very first prefetch request that installs the cache block is promoted. Therefore, under PACMAN, leslie shows significant improvement in performance. With ICP, promotion enhancement marginally improves the performance on libquantum, leslie and lbm. Since the performance enhancement on leslie is very high, PACMAN with promotion enhancement achieves similar performance

improvement on this workload and WL 1 and 2. On a similar note, PACMAN with enhanced promotion achieves higher performance on workloads such as WL 8,9,32 and 33.

5.7 Inference

In the discussions so far, we demonstrate and establish the significance of promoting the prefetched cache blocks on hits. We further showed that the state-of-the-art mechanisms could be enhanced by altering their promotion policy to treat the prefetch and demand requests alike. While altering the promotion policy enhances the performance of the two state-of-the-art mechanisms, PACMAN and ICP, we observe that the EquallyManaged policy still achieves better performance as compared to these two mechanisms. Apart from providing room for performance improvement, these two mechanisms also incur higher hardware cost. While PACMAN only requires adding one additional bit per every cache block, ICP requires additional circuitry at the level two cache for its feedback-driven prefetch-accuracy predictor. On the other hand, EquallyManaged mechanism requires no change as compared to the baseline, except investing on few counters, which is similar in overhead to PACMAN. Therefore, we can infer that the EquallyManaged mechanism, that is, treating prefetch and demand requests alike on cache insertion and promotion is better and beneficial in the context of large scale multi-core systems which employ larger caches.

5.8 Conclusion

In this chapter, we discussed the how use characteristics of prefetched cache blocks while resident at the last level cache change in the context of large scale multi-core systems. In particular, we observed that prefetched cache blocks show varying reuse behavior as compared to the zero or single reuse as observed on a single or small-scale multi-cores (2-4). Treating prefetched cache blocks as single or no reuse cache lines and not promoting their cache lines on cache hit operations becomes less effective as these policies accelerate the eviction of prefetched cache blocks. On the other hand, we observe treating the promotion of prefetched cache blocks on par with demands significantly enhances performance. We further demonstrate the significance of promotion policy on PACMAN and ICP. Though the two mechanisms improve performance, the enhancement on these two mechanisms still provide room for performance improvement. In addition to providing more room for performance improvement, PACMAN and ICP incur extra hardware cost for their implementation. The two reasons favor the baseline algorithm that learns which simply treats both demand and prefetched cache blocks alike.

Chapter 6

Conclusion and Future Work

This thesis is focused towards memory-hierarchy management in large scale multi-core processors. Along this direction, we studied managing interference in the on-chip caches and off-chip memory access. Our first work focussed on managing interference at the last level cache in the context where the number of applications sharing the cache could exceed the associativity of the shared cache. Such a scenario is possible since larger associativity leads to increased energy consumption. The second and third work focussed on managing interference at off-chip memory access and last level cache in the presence of prefetching, respectively. In the following paragraphs, we summarize our contributions as follows:

Discrete Cache Insertion Policies for large scale multi-cores

Increase in the number of applications that run on a multi-core processor increases the diversity of characteristics and memory demands the system must cater to. From the context of last level caches, applications with diverse memory behaviors share them. For efficiently utilizing the cache capacity, the cache management algorithm must take into account the diverse characteristics of applications, and accordingly prioritize them at the shared cache. Further, the need for enabling different priorities across is fueled by the fairness and performance objectives of commercial grid systems, where the memory-hierarchy is shared by applications.

In Chapter 3, we showed the drawback with mechanisms that observe the number of hits or misses to predict the reuse behavior of cache lines. Prior mechanisms approximate misses from the cache as an indicator of poor reuse behavior. However, cache blocks could be prematurely evicted from the caches due to high degree of interference from many co-running applications. Therefore, approximating reuse behavior using misses becomes ineffective. Further, we demonstrate with an example the need for differently prioritizing applications at the shared last level cache.

In order to meet such a requirement, a new mechanism that effectively captures the diverse behaviors of applications at run-time and allows the replacement policy to enforce different priorities across applications, is needed. We propose one such mechanism, Adaptive and Discrete Application PrioriTization, ADAPT replacement algorithm, for managing the shared last level caches of large scale multi-cores. In particular, we mea-

sure the working-set sizes of applications at run-time by leveraging a prior state-of-the-art mechanism. Quantitative estimate of applications' cache utility (working-set sizes) explicitly allows to enforce different priorities across applications. In particular, from experiments we statically assign priorities to applications based on their inferred footprint-number values. By comparing with the prior state-of-the-art mechanisms, we demonstrate the effectiveness of our proposed mechanism, ADAPT. We further demonstrate ADAPT in the context where the number of applications sharing the cache is greater than the associativity of the shared cache. That is, for 20-core and 24-core systems sharing the last level cache where the associativity of the cache is sixteen. Altogether, ADAPT is effective and scalable.

Band-pass Prefetching : A Prefetch-fraction driven Mechanism for Prefetcher Aggressiveness Control
Hardware prefetching is a widely used memory latency hiding mechanism. While prefetching attempts to save the cycles spent by the processor on waiting for the data, prefetcher of a core may, however, interfere with the on-demand requests of other cores. Therefore, allowing prefetchers to be aggressive may harm system fairness and performance.

To manage interference caused by prefetchers at the off-chip memory access, prior studies have proposed to control prefetcher aggressiveness at run-time. These mechanisms make dynamic throttling decisions by computing parameters that infer usefulness of prefetching and prefetcher-caused interference. Using multiple metrics (driven by their thresholds) leads to incorrect decisions because a given value of a metric does not reflect the run-time behavior of an application due to interference caused when large number of applications run on the system. While CAFFEINE normalizes prefetch-usefulness and prefetcher-caused interference into a single utility model, its method of estimating prefetch-usefulness is flawed. Essentially, the two mechanisms provide room for performance improvement.

Our mechanism is built on two fundamental observations of prefetch-usefulness and prefetch-caused interference based on the fraction of prefetch requests generated by a prefetcher. First, for a given application, fewer the prefetch requests generated, less likely that they are useful. Second, more the aggregate number of prefetch requests in the system, higher the miss-penalty on the demand misses at last level cache. Based on the two observations, we introduce the concept of prefetch-fraction, which is defined as the fraction of L2 prefetch requests the prefetcher generates for an application with respect to its total requests (demand misses, L1 and L2 prefetch requests). To infer the usefulness of prefetching to an application, we compute prefetch-fraction for each application independently at the private L2 caches. To infer interference due to a prefetcher, we compute prefetch-fraction for each application at the shared LLC-DRAM interface. Based on the inference drawn on usefulness and interference, we apply prefetcher-aggressiveness control at the private L2-LLC interface and shared LLC-DRAM interface. Altogether, the two mechanisms control the flow of prefetch requests between a range of prefetch-to-demand ratios. This is analogous to Band-pass filtering in signal processing.

Experimental results show that Band-pass prefetching is effective in addressing prefetcher-caused interference. We demonstrate the robustness of this mechanism by

studying across various workload types. Finally, our mechanism is practical. Hardware overhead is very less: only 269 bytes for a 16-core system. It does not require modifying existing cache or MSHR structures.

Reuse-aware Prefetch Management
Prefetching not only interferes with on-demand requests of other applications at the off-chip memory access, but also interferes at the shared last level cache by evicting useful cache blocks of other applications. This problem is referred to as pollution. To handle pollution, prior mechanisms treat prefetch requests as single-use or zero use cache blocks. This is based on their study of prefetch reuse characteristics at the last level cache of 1MB. That is, their study is based on a single-core system, and extend their idea to multi-core systems. However, in the context of large-scale multi-cores where the last level cache sizes are larger, the scenario becomes different.

In Chapter 5, we studied the reuse characteristics of prefetched cache blocks in the context of large-scale multi-cores and showed that not all prefetched cache blocks are pollutants; prefetched cache blocks of some applications have reuse behaviors that are on-par with their demand requests. Therefore, treating the prefetched cache blocks of such applications as zero or single-reuse cache blocks results in performance loss. In particular, the two state-of-the-art mechanisms, PACMAN and ICP do not promote and demote after the first use of the prefetched cache blocks, respectively. Doing so, accelerates the eviction of prefetched cache blocks, these mechanisms provide scope for performance improvement. Using this observation, we enhance the performance of these two mechanisms. In particular, we alter their promotion policies such that the prefetched cache blocks are allowed to stay in the caches for longer duration of time until their subsequent reuses. Basically, when there is a hit on the prefetched cache block, we promote the cache lines. Nevertheless, the two mechanisms still provide room for performance improvement, which is due to their low-priority treatment of prefetched cache blocks on insertion.

On the other hand, a mechanism that treats both prefetched and demand cache blocks on par with each other on cache insertion and promotion is able to achieve higher performance as compared to the two state-of-the-art mechanisms. We experimentally demonstrate the performance potential of equally treating prefetch and demand cache blocks and argue that this mechanism better handles prefetch requests at the shared last level caches of a large-scale multi-core system.

6.1 Perspectives

6.1.1 Managing last level caches

While this thesis explored the approach of estimating the working-set sizes of applications at run-time to gauge how well an application could utilize the cache, alternative approaches can be used to measure cache utility. One such mechanism is based on the observation of eviction-to-use distance of cache blocks. An application that exhibits high degree of reuse behavior, tends to access the cache more frequently than the others. When cache blocks of such an application is prematurely evicted from the cache,

time (in misses) between their eviction and subsequent use will be shorter. Using this observation, applications could be classified into different buckets of reuse behaviors which in turn allows to enforce different priorities across applications.

6.1.2 Prefetcher Aggressiveness Control

The flow of prefetch and demand requests between the last level cache MSHRs to the off-chip memory is analogous to flow of packets in the computer networks, with the MSHRs behaving like buffers. In particular, an outstanding miss at the last level cache sits at the LLC MSHR until that request is serviced back. The delay or the service time in processing a request and the rate at which MSHR entries are cleared depends on (i) the number of in-flight requests (that are in-transit between LLC-DRAM-LLC), (ii) the optimization techniques employed by the memory controller, and (iii) the workload behavior. In essence, the processing of requests at the MSHR follows Poisson model [Ros06].

Active queue management (AQM) is a network theory approach in which congestion in networks is relaxed through intelligent dropping of packets when the buffers are about to get full. In the context of prefetcher aggressiveness control, a prefetch request is a packet, and MSHR(s) is a buffer. Prefetch requests could be dropped in anticipation when the MSHR is about to cross a specific threshold (which could be dynamically determined). Similarly, Bufferbloat is a phenomenon in which high latency in processing of network packets results due to excessive buffering. Depending on the delay on demand requests, and its relationship with MSHR sizes, prefetch requests could be conditionally dropped. Several network theory algorithms [BZ96, SV95, Zha95, NLS07] which attempt to ensure fairness among multiple sources that transfer data packets over packet-switched-networks. As mentioned, individual requests can be treated as data packets and each application and/or its prefetcher as independent sources

Glossary

Bibliography

[AB05] Susanne Albers and Markus Büttner. Integrated prefetching and caching in single and parallel disk systems. Inf. Comput., 198(1):24–39, April 2005.

[AGIn+12] Jorge Albericio, Rubén Gran, Pablo Ibáñez, Víctor Viñals, and Jose María Llabería. ABS: A low-cost adaptive controller for prefetching in a banked shared last-level cache. ACM Trans. Archit. Code Optim., 8(4):19:1–19:20, January 2012.

[arc] Intel architecture manual, http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html.

[BC95] Jean-Loup Baer and Tien-Fu Chen. Effective hardware-based data prefetching for high-performance processors. IEEE Trans. Comput., 44(5):609–623, May 1995.

[Bie11] Christian Bienia. Benchmarking Modern Multiprocessors. PhD thesis, Princeton University, January 2011.

[BIM08] Ramazan Bitirgen, Engin Ipek, and Jose F. Martinez. Coordinated management of multiple interacting resources in chip multiprocessors: A machine learning approach. In Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 41, pages 318–329, Washington, DC, USA, 2008. IEEE Computer Society.

[BS13] Nathan Beckmann and Daniel Sanchez. Jigsaw: Scalable software-defined caches. In Proceedings of the 22Nd International Conference on Parallel Architectures and Compilation Techniques, PACT '13, pages 213–224, Piscataway, NJ, USA, 2013. IEEE Press.

[BZ96] Jon C. R. Bennett and Hui Zhang. Hierarchical packet fair queueing algorithms. In Conference Proceedings on Applications, Technologies, Architectures, and Protocols for Computer Communications, SIGCOMM '96, pages 143–156, New York, NY, USA, 1996. ACM.

[CFKL95] Pei Cao, Edward W. Felten, Anna R. Karlin, and Kai Li. A study of integrated prefetching and caching strategies. SIGMETRICS Perform. Eval. Rev., 23(1):188–197, May 1995.

[CGB+12] Mainak Chaudhuri, Jayesh Gaur, Nithiyanandan Bashyam, Sreenivas Subramoney, and Joseph Nuzman. Introducing hierarchy-awareness in replacement and bypass algorithms for last-level caches. In Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques, PACT '12, pages 293–304, New York, NY, USA, 2012. ACM.

[CMT] Intel cache monitoring technology, https://software.intel.com/en-us/blogs/2014/06/18/benefit-of-cache-monitoring.

[Cor] Corei7 processors, http://www.intel.com/content/www/us/en/processors/core/core-i7-processor.html.

[CS07] Jichuan Chang and Gurindar S. Sohi. Cooperative cache partitioning for chip multiprocessors. In Proceedings of the 21st Annual International Conference on Supercomputing, ICS '07, pages 242–252, New York, NY, USA, 2007. ACM.

[CT99] Jamison D. Collins and Dean M. Tullsen. Hardware identification of cache conflict misses. In Proceedings of the 32Nd Annual ACM/IEEE International Symposium on Microarchitecture, MICRO 32, pages 126–135, Washington, DC, USA, 1999. IEEE Computer Society.

[DDS95] Fredrik Dahlgren, Michel Dubois, and Per Stenström. Sequential hardware prefetching in shared-memory multiprocessors. IEEE Trans. Parallel Distrib. Syst., 6(7):733–746, July 1995.

[DZK+12] Nam Duong, Dali Zhao, Taesu Kim, Rosario Cammarota, Mateo Valero, and Alexander V. Veidenbaum. Improving cache management policies using dynamic reuse distances. In Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO-45, pages 389–400, Washington, DC, USA, 2012. IEEE Computer Society.

[EBSH11] David Eklov, David Black-Schaffer, and Erik Hagersten. Fast modeling of shared caches in multicore systems. In Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers, HiPEAC '11, pages 147–157, New York, NY, USA, 2011. ACM.

[ELMP11] Eiman Ebrahimi, Chang Joo Lee, Onur Mutlu, and Yale N. Patt. Prefetch-aware shared resource management for multi-core systems. In Proceedings of the 38th Annual International Symposium on Computer Architecture, ISCA '11, pages 141–152, New York, NY, USA, 2011. ACM.

[EMLP09] E. Ebrahimi, O. Mutlu, C. J. Lee, and Y. N. Patt. Coordinated control of multiple prefetchers in multi-core systems. In 2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), pages 316–326, Dec 2009.

[GAV95] Antonio González, Carlos Aliagas, and Mateo Valero. A data cache with multiple caching strategies tuned to different types of locality. In Proceedings of the 9th International Conference on Supercomputing, ICS '95, pages 338–347, New York, NY, USA, 1995. ACM.

[GST13] A. Gupta, J. Sampson, and M. Bedford Taylor. Timecube: A manycore embedded processor with interference-agnostic progress tracking. In Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIII), 2013 International Conference on, pages 227–236, July 2013.

[GW10] Hongliang Gao and Chris Wilkerson. A Dueling Segmented LRU Replacement Algorithm with Adaptive Bypassing. In Joel Emer, editor, JWAC 2010 - 1st JILP Worshop on Computer Architecture Competitions: cache replacement Championship, Saint Malo, France, June 2010.

[HL06] I. Hur and C. Lin. Memory prefetching using adaptive stream detection. In 2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06), pages 397–408, Dec 2006.

[HL09] I. Hur and C. Lin. Feedback mechanisms for improving probabilistic memory prefetching. In 2009 IEEE 15th International Symposium on High Performance Computer Architecture, pages 443–454, Feb 2009.

[IIH09] Yasuo Ishii, Mary Inaba, and Kei Hiraki. Access map pattern matching for data cache prefetch. In Proceedings of the 23rd International Conference on Supercomputing, ICS '09, pages 499–500, New York, NY, USA, 2009. ACM.

[Int16] Intel. Intel optimization manual, 2016.

[ITR] International technology roadmap for semiconductors, http://www.itrs2.net/itrs-reports.html.

[Iye04] Ravi Iyer. Cqos: A framework for enabling qos in shared caches of cmp platforms. In Proceedings of the 18th Annual International Conference on Supercomputing, ICS '04, pages 257–266, New York, NY, USA, 2004. ACM.

[JBB+15] V. Jimenez, A. Buyuktosunoglu, P. Bose, F. P. O'Connell, F. Cazorla, and M. Valero. Increasing multicore system efficiency through intelligent bandwidth shifting. In 2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA), pages 39–50, Feb 2015.

[JCMH99] Teresa L. Johnson, Daniel A. Connors, Matthew C. Merten, and Wen-mei W. Hwu. Run-time cache bypassing. IEEE Trans. Comput., 48(12):1338–1354, December 1999.

[JGC+12] Victor Jiménez, Roberto Gioiosa, Francisco J. Cazorla, Alper Buyuktosunoglu, Pradip Bose, and Francis P. O'Connell. Making data prefetch smarter: Adaptive prefetching on power7. In Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques, PACT '12, pages 137–146, New York, NY, USA, 2012. ACM.

[JHQ+08] Aamer Jaleel, William Hasenplaugh, Moinuddin Qureshi, Julien Sebot, Simon Steely, Jr., and Joel Emer. Adaptive insertion policies for managing shared caches. In Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques, PACT '08, pages 208–219, New York, NY, USA, 2008. ACM.

[Jou90] Norman P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In Proceedings of the 17th Annual International Symposium on Computer Architecture, ISCA '90, pages 364–373, New York, NY, USA, 1990. ACM.

[JTSE10] Aamer Jaleel, Kevin B. Theobald, Simon C. Steely, Jr., and Joel Emer. High performance cache replacement using re-reference interval prediction (rrip). In Proceedings of the 37th Annual International Symposium on Computer Architecture, ISCA '10, pages 60–71, New York, NY, USA, 2010. ACM.

[KKD13] George Kurian, Omer Khan, and Srinivas Devadas. The locality-aware adaptive cache coherence protocol. In Proceedings of the 40th Annual International Symposium on Computer Architecture, ISCA '13, pages 523–534, New York, NY, USA, 2013. ACM.

[KPK07] G. Keramidas, P. Petoumenos, and S. Kaxiras. Cache replacement based on reuse-distance prediction. In Computer Design, 2007. ICCD 2007. 25th International Conference on, pages 245–250, Oct 2007.

[KS08] Mazen Kharbutli and Yan Solihin. Counter-based cache replacement and bypassing algorithms. IEEE Trans. Comput., 57(4):433–447, April 2008.

[KW98] Sanjeev Kumar and Christopher Wilkerson. Exploiting spatial locality in data caches using spatial footprints. In Proceedings of the 25th Annual International Symposium on Computer Architecture, ISCA '98, pages 357–368, Washington, DC, USA, 1998. IEEE Computer Society.

[LFF01a] An-Chow Lai, Cem Fide, and Babak Falsafi. Dead-block prediction & dead-block correlating prefetchers. In Proceedings of the 28th Annual International Symposium on Computer Architecture, ISCA '01, pages 144–154, New York, NY, USA, 2001. ACM.

[LFF01b] An-Chow Lai, Cem Fide, and Babak Falsafi. Dead-block prediction & dead-block correlating prefetchers. In Proceedings of the 28th Annual International Symposium on Computer Architecture, ISCA '01, pages 144–154, New York, NY, USA, 2001. ACM.

[LFHB08] Haiming Liu, Michael Ferdman, Jaehyuk Huh, and Doug Burger. Cache bursts: A new approach for eliminating dead blocks and increasing cache efficiency. In Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 41, pages 222–233, Washington, DC, USA, 2008. IEEE Computer Society.

[LGF01] Kun Luo, J. Gummaraju, and M. Franklin. Balancing thoughput and fairness in smt processors. In Performance Analysis of Systems and Software, 2001. ISPASS. 2001 IEEE International Symposium on, pages 164–171, 2001.

[Lin01] Wi-fen Lin. Reducing dram latencies with an integrated memory hierarchy design. In Proceedings of the 7th International Symposium on High-Performance Computer Architecture, HPCA '01, pages 301–, Washington, DC, USA, 2001. IEEE Computer Society.

[LMNP08] Chang Joo Lee, Onur Mutlu, Veynu Narasiman, and Yale N. Patt. Prefetch-aware dram controllers. In Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 41, pages 200–209, Washington, DC, USA, 2008. IEEE Computer Society.

[LR02] Wei-Fen Lin and Steven K. Reinhardt. Predicting last-touch references under optimal replacement. Technical report, 2002.

[LRBP01] Wei-Fen Lin, Steven K. Reinhardt, Doug Burger, and Thomas R. Puzak. Filtering superfluous prefetches using density vectors. In 19th International Conference on Computer Design (ICCD 2001), VLSI in Computers and Processors, 23-26 September 2001, Austin, TX, USA, Proceedings, pages 124–132, 2001.

[LS11] Fang Liu and Yan Solihin. Studying the impact of hardware prefetching and bandwidth partitioning in chip-multiprocessors. In Proceedings of the ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS '11, pages 37–48, New York, NY, USA, 2011. ACM.

[LYL87] R.L. Lee, Pen-Chung Yew, and D.H. Lawrie. Data prefetching in shared memory multiprocessors. Jan 1987.

[McF92] Scott McFarling. Cache replacement with dynamic exclusion. In Proceedings of the 19th Annual International Symposium on Computer Architecture, ISCA '92, pages 191–200, New York, NY, USA, 1992. ACM.

[MGST70] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. IBM Syst. J., 9(2):78–117, June 1970.

[MM08] Onur Mutlu and Thomas Moscibroda. Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared dram systems. SIGARCH Comput. Archit. News, 36(3):63–74, June 2008.

[MRG11] R. Manikantan, K. Rajan, and R. Govindarajan. Nucache: An efficient multicore cache organization based on next-use distance. In 2011 IEEE 17th International Symposium on High Performance Computer Architecture, pages 243–253, Feb 2011.

[MRG12] R Manikantan, Kaushik Rajan, and R Govindarajan. Probabilistic shared cache management (prism). SIGARCH Comput. Archit. News, 40(3):428–439, June 2012.

[NLS07] Kyle J. Nesbit, James Laudon, and James E. Smith. Virtual private caches. In Proceedings of the 34th Annual International Symposium on Computer Architecture, ISCA '07, pages 57–68, New York, NY, USA, 2007. ACM.

[Onu] Memory systems, book chapter, http://repository.cmu.edu/ece/379/.

[OWN96] Alan V. Oppenheim, Alan S. Willsky, and S. Hamid Nawab. Signals &Amp; Systems (2Nd Ed.). Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.

[Pan16] B. Panda. SPAC:A Synergistic Prefetcher Aggressiveness controller for multi-core systems. IEEE Transactions on Computers, PP(99):1–1, 2016.

[Pat04] David A. Patterson. Latency lags bandwith. Commun. ACM, 47(10):71–75, October 2004.

[PB15] Biswabandan Panda and Shankar Balachandran. CAFFEINE: A utility-driven prefetcher aggressiveness engine for multicores. ACM Trans. Archit. Code Optim., 12(3):30:1–30:25, August 2015.

[PGG+95] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed prefetching and caching. SIGOPS Oper. Syst. Rev., 29(5):79–95, December 1995.

[PKK09] Pavlos Petoumenos, Georgios Keramidas, and Stefanos Kaxiras. Instruction-based reuse-distance prediction for effective cache management. In Proceedings of the 9th International Conference on Systems, Architectures, Modeling and Simulation, SAMOS'09, pages 49–58, Piscataway, NJ, USA, 2009. IEEE Press.

[QJP+07] Moinuddin K. Qureshi, Aamer Jaleel, Yale N. Patt, Simon C. Steely, and Joel Emer. Adaptive insertion policies for high performance caching. In Proceedings of the 34th Annual International Symposium on Computer Architecture, ISCA '07, pages 381–391, New York, NY, USA, 2007. ACM.

[QP06] Moinuddin K. Qureshi and Yale N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 39, pages 423–432, Washington, DC, USA, 2006. IEEE Computer Society.

[RDK+00] Scott Rixner, William J. Dally, Ujval J. Kapasi, Peter Mattson, and John D. Owens. Memory access scheduling. SIGARCH Comput. Archit. News, 28(2):128–138, May 2000.

[RKB+09] Brian M. Rogers, Anil Krishna, Gordon B. Bell, Ken Vu, Xiaowei Jiang, and Yan Solihin. Scaling the bandwidth wall: Challenges in and avenues for cmp scaling. In Proceedings of the 36th Annual International Symposium on Computer Architecture, ISCA '09, pages 371–382, New York, NY, USA, 2009. ACM.

[Ros06] Sheldon M. Ross. Introduction to Probability Models, Ninth Edition. Academic Press, Inc., Orlando, FL, USA, 2006.

[Sha05] A.K. Sharma. Text Book Of Correlations And Regression. Discovery Publishing House, 2005.

[SK11] Daniel Sanchez and Christos Kozyrakis. Vantage: Scalable and efficient fine-grain cache partitioning. SIGARCH Comput. Archit. News, 39(3):57–68, June 2011.

[SKP10] Derek L. Schuff, Milind Kulkarni, and Vijay S. Pai. Accelerating multicore reuse distance analysis with sampling and parallelization. In Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques, PACT '10, pages 53–64, New York, NY, USA, 2010. ACM.

[SKS+11] B. Sinharoy, R. Kalla, W. J. Starke, H. Q. Le, R. Cargnoni, J. A. Van Norstrand, B. J. Ronchetti, J. Stuecheli, J. Leenstra, G. L. Guthrie, D. Q. Nguyen, B. Blaner, C. F. Marino, E. Retter, and P. Williams. Ibm power7 multicore server processor. IBM Journal of Research and Development, 55(3):1:1–1:29, May 2011.

[SMKM12] Vivek Seshadri, Onur Mutlu, Michael A. Kozuch, and Todd C. Mowry. The evicted-address filter: A unified mechanism to address both cache pollution and thrashing. In Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques, PACT '12, pages 355–366, New York, NY, USA, 2012. ACM.

[SMKP07] S. Srinath, O. Mutlu, H. Kim, and Y. N. Patt. Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers. In 2007 IEEE 13th International Symposium on High Performance Computer Architecture, pages 63–74, Feb 2007.

[SPE] SPEC CPU benchmarks, howpublished = `https://www.spec.org/cpu/`, note = Accessed: 2016-07-17.

[STS08] Livio Soares, David Tam, and Michael Stumm. Reducing the harmful effects of last-level cache polluters with an os-level, software-only pollute buffer. In Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 41, pages 258–269, Washington, DC, USA, 2008. IEEE Computer Society.

[SV95] M. Shreedhar and George Varghese. Efficient fair queueing using deficit round robin. SIGCOMM Comput. Commun. Rev., 25(4):231–242, October 1995.

[SWA+06] Stephen Somogyi, Thomas F. Wenisch, Anastassia Ailamaki, Babak Falsafi, and Andreas Moshovos. Spatial memory streaming. In Proceedings of the 33rd Annual International Symposium on Computer Architecture, ISCA '06, pages 252–263, Washington, DC, USA, 2006. IEEE Computer Society.

[SYX+15] Vivek Seshadri, Samihan Yedkar, Hongyi Xin, Onur Mutlu, Phillip B. Gibbons, Michael A. Kozuch, and Todd C. Mowry. Mitigating prefetcher-caused pollution using informed caching policies for prefetched blocks. ACM Trans. Archit. Code Optim., 11(4):51:1–51:22, January 2015.

[TH04] Masamichi Takagi and Kei Hiraki. Inter-reference gap distribution replacement: An improved replacement algorithm for set-associative caches. In Proceedings of the 18th Annual International Conference on Supercomputing, ICS '04, pages 20–30, New York, NY, USA, 2004. ACM.

[VMS12] Ricardo A. Velasquez, Pierre Michaud, and André Seznec. BADCO: Behavioral Application-Dependent Superscalar Core Model. In SAMOS XII: International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation, Samos, Greece, July 2012.

[WAM13] Tripti S. Warrier, B. Anupama, and Madhu Mutyam. An application-aware cache replacement policy for last-level caches. In Proceedings of the 26th International Conference on Architecture of Computing Systems, ARCS'13, pages 207–219, Berlin, Heidelberg, 2013. Springer-Verlag.

[WJH+11] Carole-Jean Wu, Aamer Jaleel, Will Hasenplaugh, Margaret Martonosi, Simon C. Steely, Jr., and Joel Emer. Ship: Signature-based hit predictor for high performance caching. In Proceedings of the 44th Annual

IEEE/ACM International Symposium on Microarchitecture, MICRO-44, pages 430–441, New York, NY, USA, 2011. ACM.

[WJM+11] Carole-Jean Wu, Aamer Jaleel, Margaret Martonosi, Simon C. Steely, Jr., and Joel Emer. PACMan: Prefetch-aware cache management for high performance caching. In Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO-44, pages 442–453, New York, NY, USA, 2011. ACM.

[WM95] Wm. A. Wulf and Sally A. McKee. Hitting the memory wall: Implications of the obvious. SIGARCH Comput. Archit. News, 23(1):20–24, March 1995.

[XL09] Yuejian Xie and Gabriel H. Loh. Pipp: Promotion/insertion pseudo-partitioning of multi-core shared caches. In Proceedings of the 36th Annual International Symposium on Computer Architecture, ISCA '09, pages 174–183, New York, NY, USA, 2009. ACM.

[Zha95] Hui Zhang. Service disciplines for guaranteed performance service in packet-switching networks. Proceedings of the IEEE, 83(10):1374–1396, Oct 1995.

[ZL03] X. Zhuang and H. H. S. Lee. A hardware-based cache pollution filtering mechanism for aggressive prefetches. In Parallel Processing, 2003. Proceedings. 2003 International Conference on, pages 286–293, Oct 2003.

[ZZZ00] Zhao Zhang, Zhichun Zhu, and Xiaodong Zhang. A permutation-based page interleaving scheme to reduce row-buffer conflicts and exploit data locality. In Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture, MICRO 33, pages 32–41, New York, NY, USA, 2000. ACM.

List of Figures